4 Linear Programming

Manufacturing with Molds

Most objects we see around us today—from car bodies to plastic cups and cutlery—are made using some form of automated manufacturing. Computers play an important role in this process, both in the design phase and in the construction phase; CAD/CAM facilities are a vital part of any modern factory. The construction process used to manufacture a specific object depends on factors such as the material the object should be made of, the shape of the object, and whether the object will be mass produced. In this chapter we study some geometric aspects of manufacturing with molds, a commonly used process for plastic or metal objects. For metal objects this process is often referred to as *casting*.

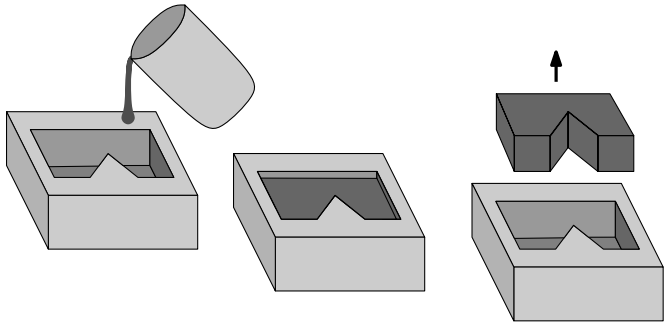


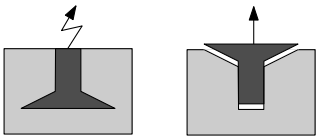
Figure 4.1
The casting process

Figure 4.1 illustrates the casting process: liquid metal is poured into a mold, it solidifies, and then the object is removed from the mold. The last step is not always as easy as it seems; the object could be stuck in the mold, so that it cannot be removed without breaking the mold. Sometimes we can get around this problem by using a different mold. There are also objects, however, for which no good mold exists; a sphere is an example. This is the problem we shall study in this chapter: given an object, is there a mold for it from which it can be removed?

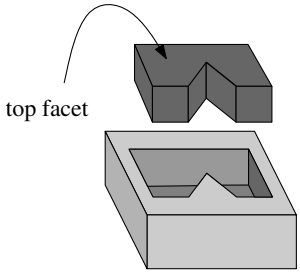
We shall confine ourselves to the following situation. First of all, we assume that the object to be constructed is polyhedral. Secondly, we only consider

molds of one piece, not molds consisting of two or more pieces. (Using molds consisting of two pieces, it is possible to manufacture objects such as spheres, which cannot be manufactured using a mold of a single piece.) Finally, we only allow the object to be removed from the mold by a single translation. This means that we will not be able to remove a screw from its mold. Fortunately, translational motions suffice for many objects.

4.1 The Geometry of Casting



If we want to determine whether an object can be manufactured by casting, we have to find a suitable mold for it. The shape of the cavity in the mold is determined by the shape of the object, but different orientations of the object give rise to different molds. Choosing the orientation can be crucial: some orientations may give rise to molds from which the object cannot be removed, while other orientations allow removal of the object. One obvious restriction on the orientation is that the object must have a horizontal *top facet*. This facet will be the only one not in contact with the mold. Hence, there are as many potential orientations—or, equivalently, possible molds—as the object has facets. We call an object *castable* if it can be removed from its mold for at least one of these orientations. In the following we shall concentrate on determining whether an object is removable by a translation from a specific given mold. To decide on the castability of the object we then simply try every potential orientation.



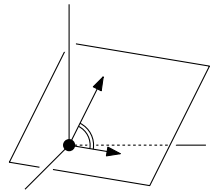
Let $\mathcal{P}$ be a 3-dimensional polyhedron—that is, a 3-dimensional solid bounded by planar facets—with a designated top facet. (We shall not try to give a precise, formal definition of a polyhedron. Giving such a definition is tricky and not necessary in this context.) We assume that the mold is a rectangular block with a cavity that corresponds exactly to $\mathcal{P}$. When the polyhedron is placed in the mold, its top facet should be coplanar with the topmost facet of the mold, which we assume to be parallel to the xy-plane. This means that the mold has no unnecessary parts sticking out on the top that might prevent $\mathcal{P}$ from being removed.

We call a facet of $\mathcal{P}$ that is not the top facet an *ordinary facet*. Every ordinary facet f has a corresponding facet in the mold, which we denote by $\hat{f}$.

We want to decide whether $\mathcal{P}$ can be removed from its mold by a single translation. In other words, we want to decide whether a direction $\vec{d}$ exists such that $\mathcal{P}$ can be translated to infinity in direction $\vec{d}$ without intersecting the interior of the mold during the translation. Note that we allow $\mathcal{P}$ to slide along the mold. Because the facet of $\mathcal{P}$ not touching the mold is its top facet, the removal direction has to be upward, that is, it must have a positive z-component. This is only a necessary condition on the removal direction; we need more constraints to be sure that a direction is valid.

Let f be an ordinary facet of $\mathcal{P}$. This facet must move away from, or slide along, its corresponding facet $\hat{f}$ of the mold. To make this constraint precise, we need to define the angle of two vectors in 3-space. We do this as follows.

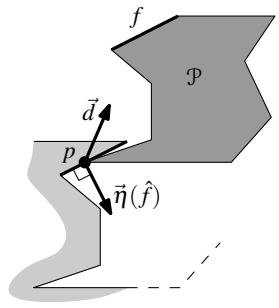
Take the plane spanned by the vectors (we assume both vectors are rooted at the origin); the angle of the vectors is the smaller of the two angles measured in this plane. Now $\hat{f}$ blocks any translation in a direction making an angle of less than 90° with $\vec{\eta}(f)$, the outward normal of f. So a necessary condition on $\vec{d}$ is that it makes an angle of at least 90° with the outward normal of every ordinary facet of $\mathcal{P}$. The next lemma shows that this condition is also sufficient.



Lemma 4.1 *The polyhedron $\mathcal{P}$ can be removed from its mold by a translation in direction $\vec{d}$ if and only if $\vec{d}$ makes an angle of at least 90° with the outward normal of all ordinary facets of $\mathcal{P}$.*

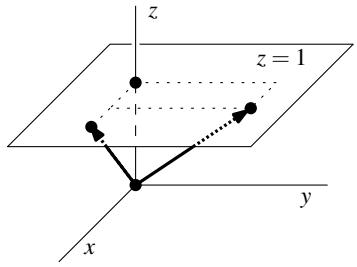
Proof. The "only if" part is easy: if $\vec{d}$ made an angle less than 90° with some outward normal $\vec{\eta}(f)$, then any point q in the interior of f collides with the mold when translated in direction $\vec{d}$.

To prove the "if" part, suppose that at some moment $\mathcal{P}$ collides with the mold when translated in direction $\vec{d}$. We have to show that there must be an outward normal making an angle of less than 90° with $\vec{d}$. Let p be a point of $\mathcal{P}$ that collides with a facet $\hat{f}$ of the mold. This means that p is about to move into the interior of the mold, so $\vec{\eta}(\hat{f})$, the outward normal of $\hat{f}$, must make an angle greater than 90° with $\vec{d}$. But then $\vec{d}$ makes an angle less than 90° with the outward normal of the ordinary facet f of $\mathcal{P}$ that corresponds to $\hat{f}$. □



Lemma 4.1 has an interesting consequence: if $\mathcal{P}$ can be removed by a sequence of small translations, then it can be removed by a single translation. So allowing for more than one translation does not help in removing the object from its mold.

We are left with the task of finding a direction $\vec{d}$ that makes an angle of at least 90° with the outward normal of each ordinary facet of $\mathcal{P}$. A direction in 3-dimensional space can be represented by a vector rooted at the origin. We already know that we can restrict our attention to directions with a positive z-component. We can represent all such directions as points in the plane $z = 1$, where the point $(x, y, 1)$ represents the direction of the vector $(x, y, 1)$. This way every point in the plane $z = 1$ represents a unique direction, and every direction with a positive z-value is represented by a unique point in that plane.

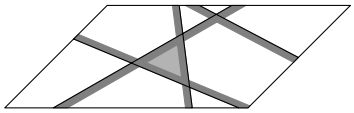


Lemma 4.1 gives necessary and sufficient conditions on the removal direction $\vec{d}$. How do these conditions translate into our plane of directions? Let $\vec{\eta} = (\vec{\eta}_x, \vec{\eta}_y, \vec{\eta}_z)$ be the outward normal of an ordinary facet. The direction $\vec{d} = (d_x, d_y, 1)$ makes an angle at least 90° with $\vec{\eta}$ if and only if the dot product of $\vec{d}$ and $\vec{\eta}$ is non-positive. Hence, an ordinary facet induces a constraint of the form

$$\vec{\eta}_x d_x + \vec{\eta}_y d_y + \vec{\eta}_z \leqslant 0.$$

This inequality describes a half-plane on the plane $z = 1$, that is, the area left or the area right of a line on the plane. (This last statement is not true for horizontal facets, which have $\vec{\eta}_x = \vec{\eta}_y = 0$. In this case the constraint is either impossible to satisfy or always satisfied, which is easy to test.) Hence, every non-horizontal facet of $\mathcal{P}$ defines a closed half-plane on the plane $z = 1$, and any point in the

common intersection of these half-planes corresponds to a direction in which $\mathcal{P}$ can be removed. The common intersection of these half-planes may be empty; in this case $\mathcal{P}$ cannot be removed from the given mold.



We have transformed our manufacturing problem to a purely geometric problem in the plane: given a set of half-planes, find a point in their common intersection or decide that the common intersection is empty. If the polyhedron to be manufactured has n facets, then the planar problem has at most $n-1$ half-planes (the top facet does not induce a half-plane). In the next sections we will see that the planar problem just stated can be solved in expected linear time—see Section 4.4, where also the meaning of "expected" is explained.

Recall that the geometric problem corresponds to testing whether $\mathcal{P}$ can be removed from a given mold. If this is impossible, there can still be other molds, corresponding to different choices of the top facet, from which $\mathcal{P}$ is removable. In order to test whether $\mathcal{P}$ is castable, we try all its facets as top facets. This leads to the following result.

Theorem 4.2 *Let $\mathcal{P}$ be a polyhedron with n facets. In $O(n^2)$ expected time and using $O(n)$ storage it can be decided whether $\mathcal{P}$ is castable. Moreover, if $\mathcal{P}$ is castable, a mold and a valid direction for removing $\mathcal{P}$ from it can be computed in the same amount of time.*

4.2 Half-Plane Intersection

Let $H = \{h_1, h_2, \ldots, h_n\}$ be a set of linear constraints in two variables, that is, constraints of the form

$$a_i x + b_i y \leqslant c_i,$$

where a_i, b_i, and c_i are constants such that at least one of a_i and b_i is non-zero. Geometrically, we can interpret such a constraint as a closed half-plane in $\mathbb{R}^2$, bounded by the line $a_i x + b_i y = c_i$. The problem we consider in this section is to find the set of all points $(x,y) \in \mathbb{R}^2$ that satisfy all n constraints at the same time. In other words, we want to find all the points lying in the common intersection of the half-planes in H. (In the previous section we reduced the casting problem to finding *some* point in the intersection of a set of half-planes. The problem we study now is more general.)

The shape of the intersection of a set of half-planes is easy to determine: a half-plane is convex, and the intersection of convex sets is again a convex set, so the intersection of a set of half-planes is a convex region in the plane. Every point on the intersection boundary must lie on the bounding line of some half-plane. Hence, the boundary of the region consists of edges contained in these bounding lines. Since the intersection is convex, every bounding line can contribute at most one edge. It follows that the intersection of n half-planes is a convex polygonal region bounded by at most n edges. Figure 4.2 shows a few examples of intersections of half-planes. To which side of its bounding

line a half-plane lies is indicated by dark shading in the figure; the common intersection is shaded lightly. As you can see in Figures 4.2 (ii) and (iii), the

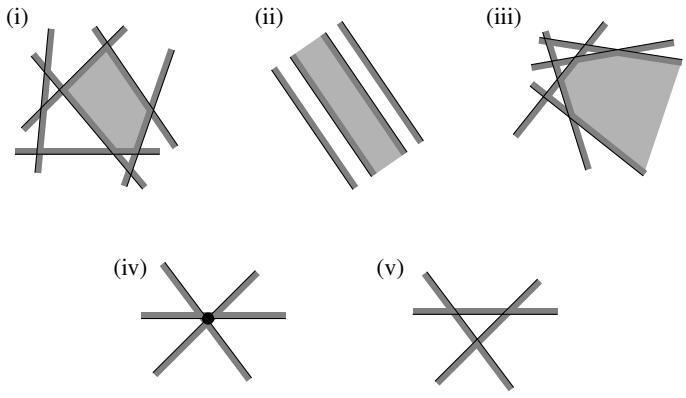


Figure 4.2
Examples of the intersection of half-planes

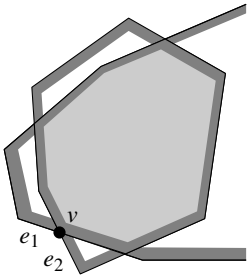
intersection does not have to be bounded. The intersection can also degenerate to a line segment or a point, as in (iv), or it can be empty, as in (v).

We give a rather straightforward divide-and-conquer algorithm to compute the intersection of a set of n half-planes. It is based on a routine INTERSECTCONVEXREGIONS to compute the intersection of two convex polygonal regions. We first give the overall algorithm.

Algorithm INTERSECTHALFPLANES(H)
Input. A set H of n half-planes in the plane.
Output. The convex polygonal region $C := \bigcap_{h \in H} h$.
1. **if** card$(H) = 1$
2. **then** $C \leftarrow$ the unique half-plane $h \in H$
3. **else** Split H into sets H_1 and H_2 of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$.
4. $C_1 \leftarrow$INTERSECTHALFPLANES(H_1)
5. $C_2 \leftarrow$INTERSECTHALFPLANES(H_2)
6. $C \leftarrow$INTERSECTCONVEXREGIONS(C_1, C_2)

What remains is to describe the procedure INTERSECTCONVEXREGIONS. But wait—didn't we see this problem before, in Chapter 2? Indeed, Corollary 2.7 states that we can compute the intersection of two polygons in $O(n \log n + k \log n)$ time, where n is the total number of vertices in the two polygons. We must be a bit careful in applying this result to our problem, because the regions we have can be unbounded, or degenerate to a segment or a point. Hence, the regions are not necessarily polygons. But it is not difficult to modify the algorithm from Chapter 2 so that it still works.

Let's analyze this approach. Assume we have already computed the two regions C_1 and C_2 by recursion. Since they are both defined by at most $n/2+1$ half-planes, they both have at most $n/2+1$ edges. The algorithm from Chapter 2 computes their overlay in time $O((n+k)\log n)$, where k is the number of intersection points between edges of C_1 and edges of C_2. What is k? Look



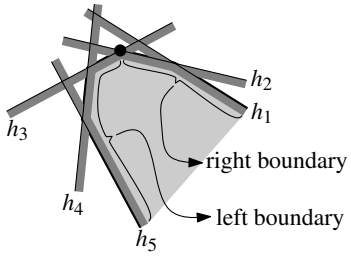
at an intersection point v between an edge e_1 of C_1 and an edge e_2 of C_2. No matter how e_1 and e_2 intersect, v must be a vertex of $C_1 \cap C_2$. But $C_1 \cap C_2$ is the intersection of n half-planes, and therefore has at most n edges and vertices. It follows that $k \leqslant n$, so the computation of the intersection of C_1 and C_2 takes $O(n \log n)$ time.

This gives the following recurrence for the total running time:

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

This recurrence solves to $T(n) = O(n \log^2 n)$.

To obtain this result we used a subroutine for computing the intersection of two arbitrary polygons. The polygonal regions we deal with in INTERSECT-HALFPLANES are always convex. Can we use this to develop a more efficient algorithm? The answer is yes, as we show next. We will assume that the regions we want to intersect are 2-dimensional; the case where one or both of them is a segment or a point is easier and left as an exercise.



$\mathcal{L}_{\text{left}}(C) = h_3, h_4, h_5$
$\mathcal{L}_{\text{right}}(C) = h_2, h_1$

First, let's specify more precisely how we represent a convex polygonal region C. We will store the left and the right boundary of C separately, as sorted lists of half-planes. The lists are sorted in the order in which the bounding lines of the half-planes occur when the (left or right) boundary is traversed from top to bottom. We denote the left boundary list by $\mathcal{L}_{\text{left}}(C)$, and the right boundary list by $\mathcal{L}_{\text{right}}(C)$. Vertices are not stored explicitly; they can be computed by intersecting consecutive bounding lines.

To simplify the description of the algorithm, we shall assume that there are no horizontal edges. (To adapt the algorithm to deal with horizontal edges, one can define such edges to belong to the left boundary if they bound C from above, and to the right boundary if they bound C from below. With this convention only a few adaptations are needed to the algorithm stated below.)

The new algorithm is a plane sweep algorithm, like the one in Chapter 2: we move a sweep line downward over the plane, and we maintain the edges of C_1 and C_2 intersecting the sweep line. Since C_1 and C_2 are convex, there are at most four such edges. Hence, there is no need to store these edges in a complicated data structure; instead we simply have pointers *left_edge_C1*, *right_edge_C1*, *left_edge_C2*, and *right_edge_C2* to them. If the sweep line does not intersect the right or left boundary of a region, then the corresponding pointer is nil. Figure 4.3 illustrates the definitions.

How are these pointers initialized? Let y_1 be the y-coordinate of the topmost vertex of C_1; if C_1 has an unbounded edge extending upward to infinity then we define $y_1 = \infty$. Define y_2 similarly for C_2, and let $y_{\text{start}} = \min(y_1, y_2)$. To compute the intersection of C_1 and C_2 we can restrict our attention to the part of the plane with y-coordinate less than or equal to y_{start}. Hence, we let the sweep line start at y_{start}, and we initialize the edges *left_edge_C1*, *right_edge_C1*, *left_edge_C2*, and *right_edge_C2* as the ones intersecting the line $y = y_{\text{start}}$.

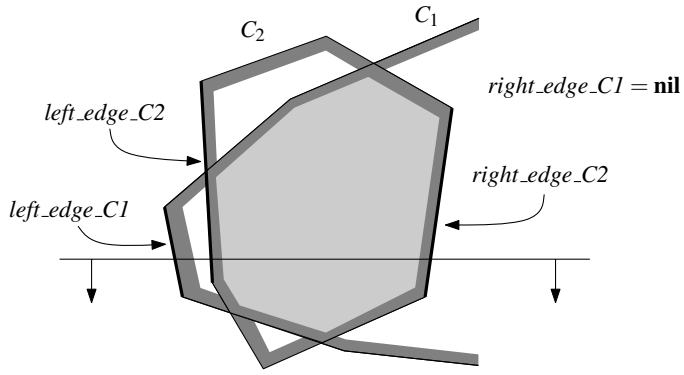


Figure 4.3
The edges maintained by the sweep line algorithm

In a plane sweep algorithm one normally also needs a queue to store the events. In our case the events are the points where edges of C_1 or of C_2 start or stop to intersect the sweep line. This implies that the next event point, which determines the next edge to be handled, is the highest of the lower endpoints of the edges intersecting the sweep line. (Endpoints with the same y-coordinate are handled from left to right. If two endpoints coincide then the leftmost edge is treated first.) Hence, we don't need an event queue; the next event can be found in constant time using the pointers *left_edge_C1*, *right_edge_C1*, *left_edge_C2*, and *right_edge_C2*.

At each event point some new edge e appears on the boundary. To handle the edge e we first check whether e belongs to C_1 or to C_2, and whether it is on the left or the right boundary, and then call the appropriate procedure. We shall only describe the procedure that is called when e is on the left boundary of C_1. The other procedures are similar.

Let p be the upper endpoint of e. The procedure that handles e will discover three possible edges that C might have: the edge with p as upper endpoint, the edge with $e \cap$ *left_edge_C2* as upper endpoint, and the edge with $e \cap$ *right_edge_C2* as upper endpoint. It performs the following actions.

- First we test whether p lies in between *left_edge_C2* and *right_edge_C2*. If this is the case, then e contributes an edge to C starting at p. We then add the half-plane whose bounding line contains e to the list $\mathcal{L}_{\text{left}}(C)$.
- Next we test whether e intersects *right_edge_C2*. If this is the case, then the intersection point is a vertex of C. Either both edges contribute an edge to C starting at the intersection point—this happens when p lies to the right of *right_edge_C2*, as in Figure 4.4(i)—or both edges contribute an edge ending there—this happens when p lies to the left of *right_edge_C2*, as in Figure 4.4(ii). If both edges contribute an edge starting at the intersection point, then we have to add the half-plane defining e to $\mathcal{L}_{\text{left}}(C)$ and the half-plane defining *right_edge_C2* to $\mathcal{L}_{\text{right}}(C)$. If they contribute an edge ending at the intersection point we do nothing; these edges have already been discovered in some other way.
- Finally we test whether e intersects *left_edge_C2*. If this is the case, then the

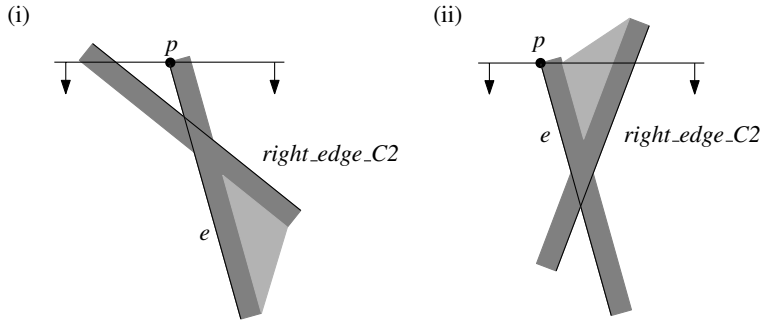
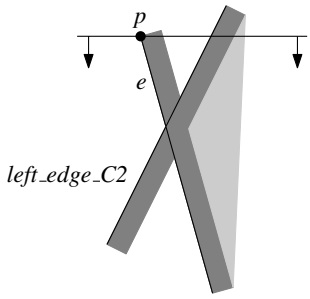


Figure 4.4
The two possibilities when e intersects *right_edge_C2*

intersection point is a vertex of C. The edge of C starting at that vertex is either a part of e or it is a part of *left_edge_C2*. We can decide between these possibilities in constant time: if p lies to the left of *left_edge_C2* then it is a part of e, otherwise it is a part of *left_edge_C2*. After we decided whether e or *left_edge_C2* contributes the edge to C, we add the appropriate half-plane to $\mathcal{L}_{\text{left}}(C)$.



Notice that we may add two half-planes to $\mathcal{L}_{\text{left}}(C)$: the half-plane bounding e and the half-plane bounding *left_edge_C2*. In which order should we add them? We add *left_edge_C2* only if it defines an edge of C starting at the intersection point of *left_edge_C2* and e. If we also decide to add the half-plane of e, it must be because e defines an edge of C starting at its upper endpoint or at its intersection point with *right_edge_C2*. In both cases we should add the half-plane bounding e first, which is guaranteed by the order of the tests given above.

We conclude that it takes constant time to handle an edge, so the intersection of two convex polygons can be computed in time $O(n)$. To show that the algorithm is correct, we have to prove that it adds the half-planes defining the edges of C in the right order. Consider an edge of C, and let p be its upper endpoint. Then p is either an upper endpoint of an edge in C_1 or C_2, or it is the intersection of two edges e and e' of C_1 and C_2, respectively. In the former case we discover the edge of C when p is reached, and in the latter case when the lower of the upper endpoints of e and e' is reached. Hence, all half-planes defining the edges of C are added. It is not difficult to prove that they are added in the correct order.

We get the following result:

Theorem 4.3 *The intersection of two convex polygonal regions in the plane can be computed in $O(n)$ time.*

This theorem shows that we can do the merge step in INTERSECTHALFPLANES in linear time. Hence, the recurrence for the running time of the algorithm becomes

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

leading to the following result:

Corollary 4.4 *The common intersection of a set of n half-planes in the plane can be computed in $O(n \log n)$ time and linear storage.*

The problem of computing the intersection of half-planes is intimately related to the computation of convex hulls, and an alternative algorithm can be given that is almost identical to algorithm CONVEXHULL from Chapter 1. The relationship between convex hulls and intersections of half-planes is discussed in detail in Sections 8.2 and 11.4. Those sections are independent of the rest of their chapters, so if you are curious you can already have a look.

4.3 Incremental Linear Programming

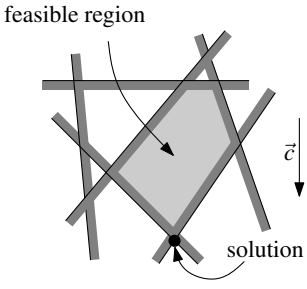
In the previous section we showed how to compute the intersection of a set of n half-planes. In other words, we computed all solutions to a set of n linear constraints. The running time of our algorithm was $O(n \log n)$. One can prove that this is optimal: as for the sorting problem, any algorithm that solves the half-plane intersection problem must take $\Omega(n \log n)$ time in the worst case. In our application to the casting problem, however, we don't need to know *all* solutions to the set of linear constraints; just one solution will do fine. It turns out that this allows for a faster algorithm.

Finding a solution to a set of linear constraints is closely related to a well-known problem in operations research, called *linear optimization* or *linear programming*. (This term was coined before "programming" came to mean "giving instructions to a computer".) The only difference is that linear programming involves finding one specific solution to the set of constraints, namely the one that maximizes a given linear function of the variables. More precisely, a linear optimization problem is described as follows:

$$\begin{aligned} \text{Maximize} \quad & c_1 x_1 + c_2 x_2 + \cdots + c_d x_d \\ \\ \text{Subject to} \quad & a_{1,1} x_1 + \cdots + a_{1,d} x_d \leqslant b_1 \\ & a_{2,1} x_1 + \cdots + a_{2,d} x_d \leqslant b_2 \\ & \qquad \vdots \\ & a_{n,1} x_1 + \cdots + a_{n,d} x_d \leqslant b_n \end{aligned}$$

where the c_i, and $a_{i,j}$, and b_i are real numbers, which form the input to the problem. The function to be maximized is called the *objective function*, and the set of constraints together with the objective function is a *linear program*. The number of variables, d, is the *dimension* of the linear program. We already saw that linear constraints can be viewed as half-spaces in $\mathbb{R}^d$. The intersection of these half-spaces, which is the set of points satisfying all constraints, is called the *feasible region* of the linear program. Points (solutions) in this region are called *feasible*, points outside are *infeasible*. Recall from Figure 4.2 that the feasible region can be unbounded, and that it can be empty. In the latter case, the linear program is called *infeasible*. The objective function can be viewed as a direction in $\mathbb{R}^d$; maximizing $c_1 x_1 + c_2 x_2 + \cdots + c_d x_d$ means finding

a point $(x_1,\ldots,x_d)$ that is extreme in the direction $\vec{c} = (c_1,\ldots,c_d)$. Hence, the solution to the linear program is a point in the feasible region that is extreme in direction $\vec{c}$. We let $f_{\vec{c}}$ denote the objective function defined by a direction vector $\vec{c}$.



Many problems in operations research can be described by linear programs, and a lot of work has been dedicated to linear optimization. This has resulted in many different linear programming algorithms, several of which—the famous *simplex algorithm* for instance—perform well in practice.

Let's go back to our problem. We have n linear constraints in two variables and we want to find one solution to the set of constraints. We can do this by taking an arbitrary objective function, and then solving the linear program defined by the objective function and the linear constraints. For the latter step we can use the simplex algorithm, or any other linear programming algorithm developed in operations research. However, this particular linear program is quite different from the ones usually studied: in operations research both the number of constraints and the number of variables are large, but in our case the number of variables is only two. The traditional linear programming methods are not very efficient in such *low-dimensional linear programming problems*; methods developed in computational geometry, like the one described below, do better.

We denote the set of n linear constraints in our 2-dimensional linear programming problem by H. The vector defining the objective function is $\vec{c} = (c_x, c_y)$; thus the objective function is $f_{\vec{c}}(p) = c_x p_x + c_y p_y$. Our goal is to find a point $p \in \mathbb{R}^2$ such that $p \in \bigcap H$ and $f_{\vec{c}}(p)$ is maximized. We denote the linear program by $(H,\vec{c})$, and we use C to denote its feasible region. We can distinguish four cases for the solution of a linear program $(H,\vec{c})$. The four cases are illustrated in Figure 4.5; the vector defining the objective function is vertically downward in the examples.

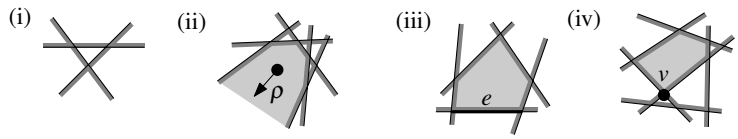


Figure 4.5
Different types of solutions to a linear program.

(i) The linear program is infeasible, that is, there is no solution to the set of constraints.

(ii) The feasible region is unbounded in direction $\vec{c}$. In this case there is a ray ρ completely contained in the feasible region C, such that the function $f_{\vec{c}}$ takes arbitrarily large values along ρ. The solution we require in this case is the description of such a ray.

(iii) The feasible region has an edge e whose outward normal points in the direction $\vec{c}$. In this case, there is a solution to the linear program, but it is not unique: any point on e is a feasible point that maximizes $f_{\vec{c}}(p)$.

(iv) If none of the preceding three cases applies, then there is a unique solution, which is the vertex v of C that is extreme in the direction $\vec{c}$.

Our algorithm for 2-dimensional linear programming is incremental. It adds the constraints one by one, and maintains the optimal solution to the intermediate linear programs. It requires, however, that the solution to each intermediate problem is well-defined and unique. In other words, it assumes that each intermediate feasible region has a unique *optimal vertex* as in case (iv) above.

To fulfill this requirement, we add to our linear program two additional constraints that will guarantee that the linear program is bounded. For example, if $c_x > 0$ and $c_y > 0$ we add the contraints $p_x \leqslant M$ and $p_y \leqslant M$, for some large $M \in \mathbb{R}$. The idea is that M should be chosen so large that the additional constraints do not influence the optimal solution, if the original linear program was bounded.

In many practical applications of linear programming, a bound of this form is actually a natural restriction. In our application to the casting problem, for instance, mechanical limitations will not allow us to remove the polyhedron in a direction that is nearly horizontal. For instance, we may not be able to remove the polyhedron in a direction whose angle with the xy-plane is less than 1 degree. This constraint immediately gives a bound on the absolute value of p_x, p_y.

We will discuss in Section 4.5 how we can correctly recognize *unbounded* linear programs, and how we can solve bounded ones without enforcing artificial constraints on the solution.

For preciseness, let's give a name to the two new constraints:

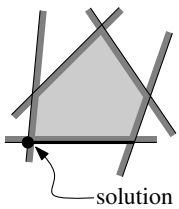
$$m_1 := \begin{cases} p_x \leqslant M & \text{if } c_x > 0 \\ -p_x \leqslant M & \text{otherwise} \end{cases}$$

and

$$m_2 := \begin{cases} p_y \leqslant M & \text{if } c_y > 0 \\ -p_y \leqslant M & \text{otherwise} \end{cases}$$

Note that m_1, m_2 are chosen as a function of $\vec{c}$ only, they do not depend on the half-planes H. The feasible region $C_0 = m_1 \cap m_2$ is an orthogonal wedge.

Another simple convention now allows us to say that case (iii) also has a unique solution: if there are several optimal points, then we want the lexicographically smallest one. Conceptually, this convention is equivalent to rotating $\vec{c}$ a little, such that it is no longer normal to any half-plane.



We have to be careful when doing this, as even a bounded linear program may not have a lexicographically smallest solution (see Exercise 4.11). Our choice of the two constraints m_1 and m_2 is such that this cannot happen.

With these two conventions, any linear program that is feasible has a unique solution, which is a vertex of the feasible region. We call this vertex the *optimal vertex*.

Let $(H, \vec{c})$ be a linear program. We number the half-planes $h_1, h_2, \ldots, h_n$. Let H_i be the set of the first i constraints, together with the special constraints m_1 and m_2, and let C_i be the feasible region defined by these constraints:

$$\begin{aligned} H_i &:= \{m_1, m_2, h_1, h_2 \ldots, h_i\}, \\ C_i &:= m_1 \cap m_2 \cap h_1 \cap h_2 \cap \cdots \cap h_i. \end{aligned}$$

By our choice of C_0, each feasible region C_i has a unique optimal vertex, denoted v_i. Clearly, we have

$$C_0 \supseteq C_1 \supseteq C_2 \cdots \supseteq C_n = C.$$

This implies that if $C_i = \emptyset$ for some i, then $C_j = \emptyset$ for all $j \geqslant i$, and the linear program is infeasible. So our algorithm can stop once the linear program becomes infeasible.

The next lemma investigates how the optimal vertex changes when we add a half-plane h_i. It is the basis of our algorithm.

Lemma 4.5 *Let $1 \leqslant i \leqslant n$, and let C_i and v_i be defined as above. Then we have*
(i) If $v_{i-1} \in h_i$, then $v_i = v_{i-1}$.
(ii) If $v_{i-1} \notin h_i$, then either $C_i = \emptyset$ or $v_i \in \ell_i$, where ℓ_i is the line bounding h_i.

Proof. (i) Let $v_{i-1} \in h_i$. Because $C_i = C_{i-1} \cap h_i$ and $v_{i-1} \in C_{i-1}$ this means that $v_{i-1} \in C_i$. Furthermore, the optimal point in C_i cannot be better than the optimal point in C_{i-1}, since $C_i \subseteq C_{i-1}$. Hence, v_{i-1} is the optimal vertex in C_i as well.

(ii) Let $v_{i-1} \notin h_i$. Suppose for a contradiction that C_i is not empty and that v_i does not lie on ℓ_i. Consider the line segment $\overline{v_{i-1}v_i}$. We have $v_{i-1} \in C_{i-1}$ and, since $C_i \subset C_{i-1}$, also $v_i \in C_{i-1}$. Together with the convexity of C_{i-1}, this implies that the segment $\overline{v_{i-1}v_i}$ is contained in C_{i-1}. Since v_{i-1} is the optimal point in C_{i-1} and the objective function $f_{\vec{c}}$ is linear, it follows that $f_{\vec{c}}(p)$ increases monotonically along $\overline{v_{i-1}v_i}$ as p moves from v_i to v_{i-1}. Now consider the intersection point q of $\overline{v_{i-1}v_i}$ and ℓ_i. This intersection point exists, because $v_{i-1} \notin h_i$ and $v_i \in C_i$. Since $\overline{v_{i-1}v_i}$ is contained in C_{i-1}, the point q must be in C_i. But the value of the objective function increases along $\overline{v_{i-1}v_i}$, so $f_{\vec{c}}(q) > f_{\vec{c}}(v_i)$. This contradicts the definition of v_i. □

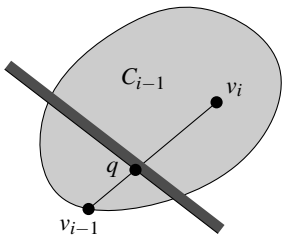


Figure 4.6 illustrates the two cases that arise when adding a half-plane. In Figure 4.6(i), the optimal vertex v_4 that we have after adding the first four half-planes is contained in h_5, the next half-plane that we add. Therefore the optimal vertex remains the same. The optimal vertex is not contained in h_6, however, so when we add h_6 we must find a new optimal vertex. According

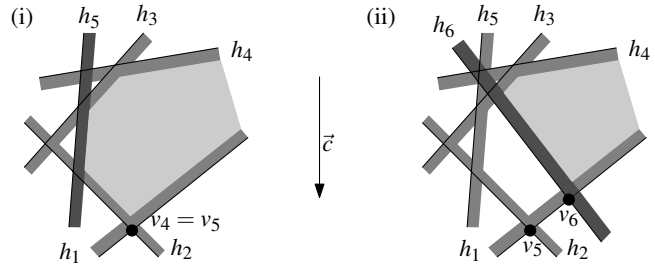


Figure 4.6
Adding a half-plane

to Lemma 4.5, this vertex v_6 is contained in the line bounding h_6, as is shown in Figure 4.6(ii). But Lemma 4.5 does not tell us how to find the new optimal vertex. Fortunately, this is not so difficult, as we show next.

Assume that the current optimal vertex v_{i-1} is not contained in the next half-plane h_i. The problem we have to solve can be stated as follows:

> Find the point p on ℓ_i that maximizes $f_{\vec{c}}(p)$, subject to the constraints $p \in h$, for $h \in H_{i-1}$.

To simplify the terminology, we assume that ℓ_i is not vertical, and so we can parameterize it by x-coordinate. We can then define a function $\overline{f_{\vec{c}}} : \mathbb{R} \mapsto \mathbb{R}$ such that $f_{\vec{c}}(p) = \overline{f_{\vec{c}}}(p_x)$ for points $p \in \ell_i$. For a half-plane h, let $\sigma(h, \ell_i)$ be the x-coordinate of the intersection point of ℓ_i and the bounding line of h. (If there is no intersection, then either the constraint h is satisfied by any point on ℓ_i, or by no point on ℓ_i. In the former case we can ignore the constraint, in the latter case we can report the linear program infeasible.) Depending on whether $\ell_i \cap h$ is bounded to the left or to the right, we get a constraint on the x-coordinate of the solution of the form $x \geqslant \sigma(h, \ell_i)$ or of the form $x \leqslant \sigma(h, \ell_i)$. We can thus restate our problem as follows:

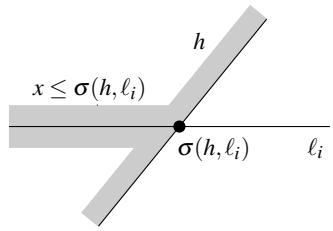
$$\begin{aligned} &\text{Maximize} && \overline{f_{\vec{c}}}(x) \\ &\text{subject to} && x \geqslant \sigma(h, \ell_i), \quad h \in H_{i-1} \text{ and } \ell_i \cap h \text{ is bounded to the left} \\ &&& x \leqslant \sigma(h, \ell_i), \quad h \in H_{i-1} \text{ and } \ell_i \cap h \text{ is bounded to the right} \end{aligned}$$

This is a 1-dimensional linear program. Solving it is very easy. Let

$$x_{\text{left}} = \max_{h \in H_{i-1}} \{\sigma(h, \ell_i) : \ \ell_i \cap h \text{ is bounded to the left}\}$$

and

$$x_{\text{right}} = \min_{h \in H_{i-1}} \{\sigma(h, \ell_i) : \ \ell_i \cap h \text{ is bounded to the right}\}.$$



The interval $[x_{\text{left}} : x_{\text{right}}]$ is the feasible region of the 1-dimensional linear program. Hence, the linear program is infeasible if $x_{\text{left}} > x_{\text{right}}$, and otherwise the optimal point is the point on ℓ_i at either x_{left} or x_{right}, depending on the objective function.

Note that the 1-dimensional linear program cannot be unbounded, due to the constraints m_1 and m_2.

We get the following lemma:

Lemma 4.6 *A 1-dimensional linear program can be solved in linear time. Hence, if case (ii) of Lemma 4.5 arises, then we can compute the new optimal vertex v_i, or decide that the linear program is infeasible, in $O(i)$ time.*

We can now describe the linear programming algorithm in more detail. As above, we use ℓ_i to denote the line that bounds the half-plane h_i.

Algorithm 2DBOUNDEDLP($H, \vec{c}, m_1, m_2$)
Input. A linear program $(H \cup \{m_1, m_2\}, \vec{c})$, where H is a set of n half-planes, $\vec{c} \in \mathbb{R}^2$, and m_1, m_2 bound the solution.
Output. If $(H \cup \{m_1, m_2\}, \vec{c})$ is infeasible, then this fact is reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

```
1.  Let v_0 be the corner of C_0.
2.  Let h_1, ..., h_n be the half-planes of H.
3.  for i ← 1 to n
4.      do if v_{i-1} ∈ h_i
5.           then v_i ← v_{i-1}
6.           else v_i ←the point p on ℓ_i that maximizes f_c(p), subject to the
                      constraints in H_{i-1}.
7.                if p does not exist
8.                  then Report that the linear program is infeasible and quit.
9.  return v_n
```

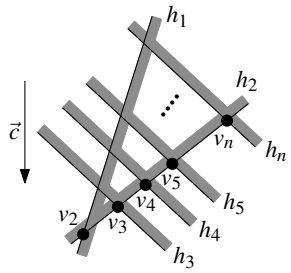
We now analyze the performance of our algorithm.

Lemma 4.7 *Algorithm* 2DBOUNDEDLP *computes the solution to a bounded linear program with n constraints and two variables in $O(n^2)$ time and linear storage.*

Proof. To prove that the algorithm correctly finds the solution, we have to show that after every stage—whenever we have added a new half-plane h_i—the point v_i is still the optimum point for C_i. This follows immediately from Lemma 4.5. If the 1-dimensional linear program on ℓ_i is infeasible, then C_i is empty, and consequently $C = C_n \subseteq C_i$ is empty, which means that the linear program is infeasible.

It is easy to see that the algorithm requires only linear storage. We add the half-planes one by one in n stages. The time spent in stage i is dominated by the time to solve a 1-dimensional linear program in line 6, which is $O(i)$. Hence, the total time needed is bounded by

$$\sum_{i=1}^{n} O(i) = O(n^2). \qquad \square$$



Although our linear programming algorithm is nice and simple, its running time is disappointing—the algorithm is much slower than the previous algorithm, which computed the whole feasible region. Is our analysis too crude? We bounded the cost of every stage i by $O(i)$. This is not always a tight bound: Stage i takes $\Theta(i)$ time only when $v_{i-1} \notin h_i$; when $v_{i-1} \in h_i$ then stage i takes constant time. So if we could bound the number of times the optimal vertex changes, we might be able to prove a better running time. Unfortunately the optimum vertex can change n times: there are orders for some configurations where every new half-plane makes the previous optimum illegal. The figure in the margin shows such an example. This means that the algorithm will really spend $\Theta(n^2)$ time. How can we avoid this nasty situation?

4.4 Randomized Linear Programming

If we have a second look at the example where the optimum changes n times, we see that the problem is not so much that the set of half-planes is bad. If we

had added them in the order $h_n, h_{n-1}, \ldots, h_3$, then the optimal vertex would not change anymore after the addition of h_n. In this case the running time would be $O(n)$. Is this a general phenomenon? Is it true that, for any set H of half-planes, there is a good order to treat them? The answer to this question is "yes," but that doesn't seem to help us much. Even if such a good order exists, there seems to be no easy way to actually *find* it. Remember that we have to find the order at the beginning of the algorithm, when we don't know anything about the intersection of the half-planes yet.

We now meet a quite intriguing phenomenon. Although we have no way to determine an ordering of H that is guaranteed to lead to a good running time, we have a very simple way out of our problem. We simply pick a *random* ordering of H. Of course, we could have bad luck and pick an order that leads to a quadratic running time. But with some luck, we pick an order that makes it run much faster. Indeed, we shall prove below that most orders lead to a fast algorithm. For completeness, we first repeat the algorithm.

Algorithm 2DRANDOMIZEDBOUNDEDLP($H, \vec{c}, m_1, m_2$)
Input. A linear program $(H \cup \{m_1, m_2\}, \vec{c})$, where H is a set of n half-planes, $\vec{c} \in \mathbb{R}^2$, and m_1, m_2 bound the solution.
Output. If $(H \cup \{m_1, m_2\}, \vec{c})$ is infeasible, then this fact is reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

```
1.  Let v_0 be the corner of C_0.
2.  Compute a random permutation h_1, ..., h_n of the half-planes by calling
    RANDOMPERMUTATION(H[1 ··· n]).
3.  for i ← 1 to n
4.      do if v_{i-1} ∈ h_i
5.           then v_i ← v_{i-1}
6.           else v_i ←the point p on ℓ_i that maximizes f_c(p), subject to the
                  constraints in H_{i-1}.
7.                if p does not exist
8.                  then Report that the linear program is infeasible and quit.
9.  return v_n
```

The only difference from the previous algorithm is in line 2, where we put the half-planes in random order before we start adding them one by one. To be able to do this, we assume that we have a random number generator, RANDOM(k), which has an integer k as input and generates a random integer between 1 and k in constant time. Computing a random permutation can then be done with the following linear time algorithm.

Algorithm RANDOMPERMUTATION(A)
Input. An array $A[1 \cdots n]$.
Output. The array $A[1 \cdots n]$ with the same elements, but rearranged into a random permutation.

```
1.  for k ← n downto 2
2.      do rndindex ←RANDOM(k)
3.         Exchange A[k] and A[rndindex].
```

The new linear programming algorithm is called a *randomized algorithm*; its running time depends on certain random choices made by the algorithm. (In the linear programming algorithm, these random choices were made in the subroutine RANDOMPERMUTATION.)

What is the running time of this randomized version of our incremental linear programming algorithm? There is no easy answer to that. It all depends on the order that is computed in line 2. Consider a fixed set H of n half-planes. 2DRANDOMIZEDBOUNDEDLP treats them depending on the permutation chosen in line 2. Since there are $n!$ possible permutations of n objects, there are $n!$ possible ways in which the algorithm can proceed, each with its own running time. Because the permutation is random, each of these running times is equally likely. So what we do is analyze the *expected running time* of the algorithm, which is the *average running time over all $n!$ possible permutations*. The lemma below states that the expected running time of our randomized linear programming algorithm is $O(n)$. It is important to realize that we do not make any assumptions about the input: the expectancy is with respect to the random order in which the half-planes are treated and holds for any set of half-planes.

Lemma 4.8 *The 2-dimensional linear programming problem with n constraints can be solved in $O(n)$ randomized expected time using worst-case linear storage.*

Proof. As we observed before, the storage needed by the algorithm is linear.

The running time RANDOMPERMUTATION is $O(n)$, so what remains is to analyze the time needed to add the half-planes $h_1, \ldots, h_n$. Adding a half-plane takes constant time when the optimal vertex does not change. When the optimal vertex does change we need to solve a 1-dimensional linear program. We now bound the time needed for all these 1-dimensional linear programs.

Let X_i be a random variable, which is 1 if $v_{i-1} \notin h_i$, and 0 otherwise. Recall that a 1-dimensional linear program on i constraints can be solved in $O(i)$ time. The total time spent in line 6 over all half-planes $h_1, \ldots, h_n$ is therefore

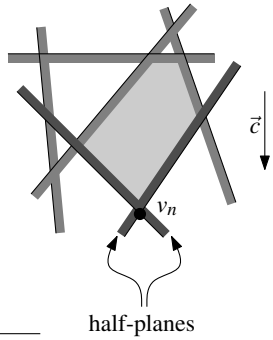
$$\sum_{i=1}^{n} O(i) \cdot X_i.$$

To bound the expected value of this sum we will use *linearity of expectation*: the expected value of a sum of random variables is the sum of the expected values of the random variables. This holds even if the random variables are dependent. Hence, the expected time for solving all 1-dimensional linear programs is

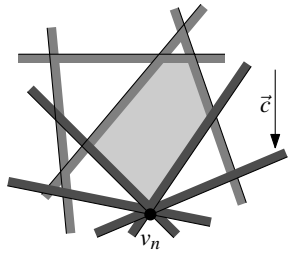
$$\mathrm{E}[\sum_{i=1}^{n} O(i) \cdot X_i] = \sum_{i=1}^{n} O(i) \cdot \mathrm{E}[X_i].$$

But what is $\mathrm{E}[X_i]$? It is exactly the probability that $v_{i-1} \notin h_i$. Let's analyze this probability.

We will do this with a technique called *backwards analysis*: we look at the algorithm "backwards." Assume that it has already finished, and that it has computed the optimum vertex v_n. Since v_n is a vertex of C_n, it is defined by at



half-planes defining v_n



least two of the half-planes. Now we make one step backwards in time, and look at C_{n-1}. Note that C_{n-1} is obtained from C_n by removing the half-plane h_n. When does the optimum point change? This happens exactly if v_n is not a vertex of C_{n-1} that is extreme in the direction $\vec{c}$, which is only possible if h_n is one of the half-planes that define v_n. But the half-planes are added in random order, so h_n is a random element of $\{h_1, h_2, \ldots, h_n\}$. Hence, the probability that h_n is one of the half-planes defining v_n is at most $2/n$. Why do we say "at most"? First, it is possible that the boundaries of more than two half-planes pass through v_n. In that case, removing one of the two half-planes containing the edges incident to v_n may fail to change v_n. Furthermore, v_n may be defined by m_1 or m_2, which are not among the n candidates for the random choice of h_n. In both cases the probability is less than $2/n$.

The same argument works in general: to bound $\mathrm{E}[X_i]$, we fix the subset of the first i half-planes. This determines C_i. To analyze what happened in the last step, when we added h_i, we think backwards. The probability that we had to compute a new optimal vertex when adding h_i is the same as the probability that the optimal vertex changes when we remove a half-plane from C_i. The latter event only takes place for at most two half-planes of our fixed set $\{h_1, \ldots, h_i\}$. Since the half-planes are added in random order, the probability that h_i is one of the special half-planes is at most $2/i$. We derived this probability under the condition that the first i half-planes are some fixed subset of H. But since the derived bound holds for *any* fixed subset, it holds unconditionally. Hence, $\mathrm{E}[X_i] \leqslant 2/i$. We can now bound the expected total time for solving all 1-dimensional linear programs by

$$\sum_{i=1}^{n} O(i) \cdot \frac{2}{i} = O(n).$$

We already noted that the time spent in the rest of the algorithm is $O(n)$ as well. □

Note again that the expectancy here is solely with respect to the random choices made by the algorithm. We do not average over possible choices for the input. For *any* input set of n half-planes, the expected running time of the algorithm is $O(n)$; there are no bad inputs.

4.5 Unbounded Linear Programs

In the preceding sections we avoided handling the case of an unbounded linear program by adding two additional, artificial constraints. This is not always a suitable solution. Even if the linear program is bounded, we may not know a large enough bound M. Furthermore, unbounded linear programs do occur in practice, and we have to solve them correctly.

Let's first see how we can recognize whether a given linear program $(H, \vec{c})$ is unbounded. As we saw before, that means that there is a ray ρ completely

contained in the feasible region C, such that the function $f_{\vec{c}}$ takes arbitrarily large values along ρ.

If we denote the ray's starting point as p, and its direction vector as $\vec{d}$, we can parameterize ρ as follows:

$$\rho = \{p + \lambda \vec{d} : \lambda > 0\}.$$

The function $f_{\vec{c}}$ takes arbitrarily large values if and only if $\vec{d} \cdot \vec{c} > 0$. On the other hand, if $\vec{\eta}(h)$ is the normal vector of a half-plane $h \in H$ oriented towards the feasible side of h's bounding line, we have $\vec{d} \cdot \vec{\eta}(h) \geqslant 0$. The next lemma shows that these two necessary conditions on $\vec{d}$ are sufficient to test whether a linear program is unbounded.

Lemma 4.9 *A linear program $(H, \vec{c})$ is unbounded if and only if there is a vector $\vec{d}$ with $\vec{d} \cdot \vec{c} > 0$ such that $\vec{d} \cdot \vec{\eta}(h) \geqslant 0$ for all $h \in H$ and the linear program $(H', \vec{c})$ is feasible, where $H' = \{h \in H : \vec{\eta}(h) \cdot \vec{d} = 0\}$.*

Proof. The "only if" direction follows from the argument above, so it remains to show the "if" direction.

We consider a linear program $(H, \vec{c})$ and a vector $\vec{d}$ with the conditions of the lemma. Since $(H', \vec{c})$ is feasible, there is a point $p_0 \in \bigcap_{h \in H'} h$. Consider now the ray $\rho_0 := \{p_0 + \lambda \vec{d} : \lambda > 0\}$. Since $\vec{d} \cdot \vec{\eta}(h) = 0$ for $h \in H'$, the ray ρ_0 is completely contained in each $h \in H'$. Furthermore, since $\vec{d} \cdot \vec{c} > 0$ the objective function $f_{\vec{c}}$ takes arbitrarily large values along ρ_0.

For a half-plane $h \in H \setminus H'$, we have $\vec{d} \cdot \vec{\eta}(h) > 0$. This implies that there is a parameter λ_h such that $p_0 + \lambda \vec{d} \in h$ for all $\lambda \geqslant \lambda_h$. Let $\lambda' := \max_{h \in H \setminus H'} \lambda_h$, and $p := p_0 + \lambda' \vec{d}$. It follows that the ray

$$\rho = \{p + \lambda \vec{d} : \lambda > 0\}$$

is completely contained in each half-plane $h \in H$, and so $(H, \vec{c})$ is unbounded. □

We can now test whether a given 2-dimensional linear program $(H, \vec{c})$ is unbounded by proceeding similarly to Section 4.1, and solving a 1-dimensional linear program.

Let's first rotate the coordinate system so that $\vec{c}$ is the upward vertical direction, $\vec{c} = (0, 1)$. Any direction vector $\vec{d} = (d_x, d_y)$ with $\vec{d} \cdot \vec{c} > 0$ can be normalized to the form $\vec{d} = (d_x, 1)$, and be represented by the point d_x on the line $y = 1$. Given a normal vector $\vec{\eta}(h) = (\eta_x, \eta_y)$, the inequality

$$\vec{d} \cdot \vec{\eta}(h) = d_x \eta_x + \eta_y \geqslant 0$$

translates to the inequality $d_x \eta_x \geqslant -\eta_y$. We thus obtain a system of n linear inequalities, or, in other words, a 1-dimensional linear program $\overline{H}$. (This is actually an abuse of the terminology, since a linear program consists of constraints and an objective function. But since at this point we are only interested in feasibility, it is convenient to ignore the objective function.)

If $\overline{H}$ has a feasible solution d_x^*, we identify the set $H' \subseteq H$ of half-planes h for which the solution is tight, that is, where $d_x^* \eta_x + \eta_y = 0$. We still need to verify that the system H' is feasible. Are we again left with a 2-dimensional linear programming problem? Yes, but a very special one: For each $h \in H'$ the normal $\vec{\eta}(h)$ is orthogonal to $\vec{d} = (d_x^*, 1)$, and that means that the bounding line of h is parallel to $\vec{d}$. In other words, all half-planes in H' are bounded by parallel lines, and by intersecting them with the x-axis, we have again a 1-dimensional linear program $\overline{H'}$. If $\overline{H'}$ is feasible, then the original linear program is unbounded, and we can construct a feasible ray ρ in time $O(n)$ as in the lemma above. If $\overline{H'}$ is infeasible, then so is H' and therefore H.

If $\overline{H}$ does not have a feasible solution, by the lemma above the original linear program $(H, \vec{c})$ is bounded. Can we extract some more information in this case? Recall the solution for 1-dimensional linear programs: $\overline{H}$ is infeasible if and only if the maximum boundary of a half-line $\overline{h_1}$ bounded to the left is larger than the minimum boundary of a half-line $\overline{h_2}$ bounded to the right. These two half-lines $\overline{h_1}$ and $\overline{h_2}$ have an empty intersection. If h_1 and h_2 are the original half-planes that correspond to these two constraints, then this is equivalent to saying that $(\{h_1, h_2\}, \vec{c})$ is bounded. We can call h_1 and h_2 *certificates*: they 'prove' that $(H, \vec{c})$ is really bounded.

How useful certificates are becomes clear with the following observation: After finding the two certificates h_1 and h_2, we can use them like m_1 and m_2 in 2DRANDOMIZEDBOUNDEDLP. That means that we no longer need to make an artificial restriction on the range in which we allow the solution to lie.

Again, we must be careful. It can happen that the linear program $(\{h_1, h_2\}, \vec{c})$ is bounded, but has no lexicographically smallest solution. This is the case when the 1-dimensional linear program is infeasible due to a single constraint h_1, namely when $\vec{\eta}(h_1) = -\vec{c} = (0, -1)$. In that case we scan the remaining list of half-planes for a half-plane h_2 with $\eta_x(h_2) > 0$. If we are successful, h_1 and h_2 are certificates that guarantee a unique lexicographically smallest solution. If no such h_2 exists, the linear program is either infeasible, or it has no lexicographically smallest solution. We can solve it by solving the 1-dimensional linear program formed by all half-planes h with $\vec{\eta}_x(h) = 0$. If it is feasible, we can return a ray ρ in direction $(-1, 0)$, such that all points on ρ are feasible optimal solutions.

We can now give a general algorithm for the 2-dimensional linear programming problem:

Algorithm 2DRANDOMIZEDLP$(H, \vec{c})$
Input. A linear program $(H, \vec{c})$, where H is a set of n half-planes and $\vec{c} \in \mathbb{R}^2$.
Output. If $(H, \vec{c})$ is unbounded, a ray is reported. If it is infeasible, then two or three certificate half-planes are reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

1. Determine whether there is a direction vector $\vec{d}$ such that $\vec{d} \cdot \vec{c} > 0$ and $\vec{d} \cdot \vec{\eta}(h) \geqslant 0$ for all $h \in H$.
2. **if** $\vec{d}$ exists
3. **then** compute H' and determine whether H' is feasible.
4. **if** H' is feasible

```
5.          then Report a ray proving that (H, c⃗) is unbounded and quit.
6.          else Report that (H, c⃗) is infeasible and quit.
7.  Let h_1, h_2 ∈ H be certificates proving that (H, c⃗) is bounded and has a
    unique lexicographically smallest solution.
8.  Let v_2 be the intersection of ℓ_1 and ℓ_2.
9.  Let h_3, h_4, ..., h_n be a random permutation of the remaining half-planes in
    H.
10. for i ← 3 to n
11.     do if v_{i−1} ∈ h_i
12.          then v_i ← v_{i−1}
13.          else v_i ←the point p on ℓ_i that maximizes f_c⃗(p), subject to the
                  constraints in H_{i−1}.
14.               if p does not exist
15.                 then Let h_j, h_k (with j, k < i) be the certificates (possibly
                         h_j = h_k) with h_j ∩ h_k ∩ ℓ_i = ∅.
16.                      Report that the linear program is infeasible, with
                         h_i, h_j, h_k as certificates, and quit.
17. return v_n
```

We summarize our results so far in the following theorem.

Theorem 4.10 *A 2-dimensional linear programming problem with n constraints can be solved in $O(n)$ randomized expected time using worst-case linear storage.*

4.6* Linear Programming in Higher Dimensions

The linear programming algorithm presented in the previous sections can be generalized to higher dimensions. When the dimension is not too high, then the resulting algorithm compares favorably with traditional algorithms, such as the simplex algorithm.

Let H be a set of n closed half-spaces in $\mathbb{R}^d$. Given a vector $\vec{c} = (c_1, \ldots, c_d)$, we want to find the point $p = (p_1, \ldots, p_d) \in \mathbb{R}^d$ that maximizes the linear function $f_{\vec{c}}(p) := c_1 p_1 + \cdots + c_d p_d$, subject to the constraint that p lies in h for all $h \in H$. To make sure that the solution is unique when the linear program is bounded, we agree to look for the lexicographically smallest point that maximizes $f_{\vec{c}}(p)$.

As in the planar version, we maintain the optimal solution while incrementally adding the half-space constraints one by one. For this to work, we again need to make sure that there is a unique optimal solution at each step. We do this as in the previous section: We first determine whether the linear program is unbounded. If not, we obtain a set of d certificates $h_1, h_2, \ldots, h_d \in H$ that guarantee that the solution is bounded and that there is a unique lexicographically smallest solution. We'll look at the details of finding these certificates later, and concentrate on the main algorithm for the moment.

Let $h_1, h_2, \ldots, h_d$ be the d certificate half-spaces obtained by checking that the linear program is bounded, and let $h_{d+1}, h_{d+2}, \ldots, h_n$ be a random permutation of the remaining half-spaces in H. Furthermore, define C_i to be the feasible

region when the first i half-spaces have been added, for $d \leqslant i \leqslant n$:

$$C_i := h_1 \cap h_2 \cap \cdots \cap h_i.$$

Let v_i denote the optimal vertex of C_i, that is, the vertex that maximizes $f_{\vec{c}}$. Lemma 4.5 gave us an easy way to maintain the optimal vertex in the 2-dimensional case: either the optimal vertex doesn't change, or the new optimal vertex is contained in the line that bounds the half-plane h_i that we are adding. The following lemma generalizes this result to higher dimensions; its proof is a straightforward generalization of the proof of Lemma 4.5.

Lemma 4.11 *Let $1 \leqslant i \leqslant n$, and let C_i and v_i be defined as above. Then we have*
(i) If $v_{i-1} \in h_i$, then $v_i = v_{i-1}$.
(ii) If $v_{i-1} \notin h_i$, then either $C_i = \emptyset$ or $v_i \in g_i$, where g_i is the hyperplane that bounds h_i.

If we denote the hyperplane that bounds the half-space h_i by g_i, the optimal vertex v_i of C_i can be found by finding the optimal vertex of the intersection $g_i \cap C_{i-1}$.

But how do we find the optimal vertex of $g_i \cap C_{i-1}$? In two dimensions this was easy to do in linear time, because everything was restricted to a line. Let's look at the 3-dimensional case. In three dimensions, g_i is a plane, and $g_i \cap C_{i-1}$ is a 2-dimensional convex polygonal region. What do we have to do to find the optimum in $g_i \cap C_{i-1}$? We have to solve a 2-dimensional linear program! The linear function $f_{\vec{c}}$ defined in $\mathbb{R}^3$ induces a linear function in g_i, and we need to find the point in $g_i \cap C_{i-1}$ that maximizes this function. In case $\vec{c}$ is orthogonal to g_i, all points on g_i are equally good: following our rule, we then need to find the lexicographically smallest solution. We achieve this by choosing the objective function correctly—for instance, when g_i is not orthogonal to the x_1-axis, we obtain the vector $\vec{c}$ by projecting the vector $(-1,0,0)$ onto g_i.

So in the 3-dimensional case we find the optimal vertex of $g_i \cap C_{i-1}$ as follows: we compute the intersection of all $i-1$ half-spaces with g_i, and project the vectors

$$\vec{c}, \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

on g_i until a projection is non-zero. This results in a linear program in two dimensions, which we solve using algorithm 2DRANDOMIZEDLP.

By now you can probably guess how we will attack the general, d-dimensional case. There, g_i is a hyperplane, a $(d-1)$-dimensional subspace, and we have to find the point in the intersection $C_{i-1} \cap g_i$ that maximizes $f_{\vec{c}}$. This is a linear program in $d-1$ dimensions, and so we will solve it by making a recursive call to the $(d-1)$-dimensional version of our algorithm. The recursion bottoms out when we get to a 1-dimensional linear program, which can be solved directly in linear time.

We still need to determine whether the linear program is unbounded, and to find suitable certificates if that is not the case. We first verify that Lemma 4.9

holds in arbitrary dimensions. The lemma and its proof need no change. The lemma implies that the d-dimensional linear program $(H, \vec{c})$ is bounded if and only if a certain $(d-1)$-dimensional linear program is infeasible. We will solve this $(d-1)$-dimensional linear program by a recursive call.

If the $(d-1)$-dimensional linear program is feasible, we obtain a direction vector $\vec{d}$. The d-dimensional linear program is then either unbounded in direction $\vec{d}$, or infeasible. This can be determined by verifying whether $(H', \vec{c})$ is feasible, where H' is as defined in Lemma 4.9. The boundaries of all the half-spaces in H' are parallel to $\vec{d}$, and so this can be decided by solving a second $(d-1)$-dimensional program, with a second recursive call.

If the $(d-1)$-dimensional linear program is infeasible, its solution will give us k certificate half-spaces $h_1, h_2, \ldots, h_k \in H$, with $k < d$, that 'prove' that $(H, \vec{c})$ is bounded. If $k < d$, then the set of optimal solutions to $(\{h_1, \ldots, h_k\}, \vec{c})$ is unbounded. In that case, these optimal solutions form a $(d-k)$-dimensional subspace. We determine whether the linear program restricted to this subspace is bounded with respect to the lexicographical order. If not, we can report the solution, otherwise we can repeat the process until we obtain a set of d certificates with a unique solution.

The global algorithm now looks as follows. Again we use g_i to denote the hyperplane that bounds the half-space h_i.

Algorithm RANDOMIZEDLP$(H, \vec{c})$
Input. A linear program $(H, \vec{c})$, where H is a set of n half-spaces in $\mathbb{R}^d$ and $\vec{c} \in \mathbb{R}^d$.
Output. If $(H, \vec{c})$ is unbounded, a ray is reported. If it is infeasible, then at most $d+1$ certificate half-planes are reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

1. Determine whether a direction vector $\vec{d}$ exists such that $\vec{d} \cdot \vec{c} > 0$ and $\vec{d} \cdot \vec{\eta}(h) \geqslant 0$ for all $h \in H$.
2. **if** $\vec{d}$ exists
3. **then** compute H' and determine whether H' is feasible.
4. **if** H' is feasible
5. **then** Report a ray proving that $(H, \vec{c})$ is unbounded and quit.
6. **else** Report that $(H, \vec{c})$ is infeasible, provide certificates, and quit.
7. Let $h_1, h_2, \ldots, h_d$ be certificates proving that $(H, \vec{c})$ is bounded.
8. Let v_d be the intersection of $g_1, g_2, \ldots, g_d$.
9. Compute a random permutation $h_{d+1}, \ldots, h_n$ of the remaining half-spaces in H.
10. **for** $i \leftarrow d+1$ **to** n
11. **do if** $v_{i-1} \in h_i$
12. **then** $v_i \leftarrow v_{i-1}$
13. **else** $v_i \leftarrow$the point p on g_i that maximizes $f_{\vec{c}}(p)$, subject to the constraints $\{h_1, \ldots, h_{i-1}\}$
14. **if** p does not exist
15. **then** Let H^* be the at most d certificates for the infeasibility of the $(d-1)$-dimensional program.

16. Report that the linear program is infeasible, with $H^* \cup h_i$ as certificates, and quit.
17. **return** v_n

The following theorem states the performance of RANDOMIZEDLP. Although we consider d a constant, which means we can state an $O(n)$ bound on the running time, it is useful to have a close look at the dependency of the running time on d—see the end of the proof the following theorem.

Theorem 4.12 *For each fixed dimension d, a d-dimensional linear programming problem with n constraints can be solved in $O(n)$ expected time.*

Proof. We must prove that there is a constant C_d such that the algorithm takes at most $C_d n$ expected time. We proceed by induction on the dimension d. For two dimensions, the result follows from Theorem 4.10, so let's assume $d > 2$. The induction step is basically identical to the proof of the 2-dimensional cases.

We start by solving at most d linear programs of dimension $d-1$. By the induction assumption, this takes time $O(dn) + dC_{d-1}n$.

The algorithm spends $O(d)$ time to compute v_d. Testing whether $v_{i-1} \in h_i$ takes $O(d)$ time. The running time is therefore $O(dn)$ as long as we do not count the time spent in line 13.

In line 13, we need to project $\vec{c}$ on g_i, in time $O(d)$, and to intersect i half-spaces with g_i, in time $O(di)$. Furthermore, we make a recursive call with dimension $d-1$ and $i-1$ half-spaces.

Define a random variable X_i, which is 1 if $v_{i-1} \notin h_i$, and 0 otherwise. The total expected time spent by the algorithm is bounded by

$$O(dn) + dC_{d-1}n + \sum_{i=d+1}^{n} (O(di) + C_{d-1}(i-1)) \cdot \mathrm{E}[X_i].$$

To bound $\mathrm{E}[X_i]$, we apply backwards analysis. Consider the situation after adding $h_1, \ldots, h_i$. The optimum point is a vertex v_i of C_i, so it is defined by d of the half-spaces. Now we make one step backwards in time. The optimum point changes only if we remove one of the half-spaces defining v_i. Since $h_{d+1}, \ldots, h_i$ is a random permutation, the probability that this happens is at most $d/(i-d)$.

Consequently, we get the following bound for the expected running time of the algorithm:

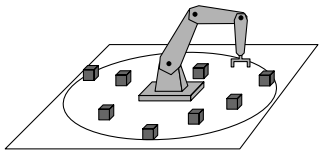
$$O(dn) + dC_{d-1}n + \sum_{i=d+1}^{n} (O(di) + C_{d-1}(i-1)) \frac{d}{i-d}$$

This can be bounded by $C_d n$, with $C_d = O(C_{d-1}d)$, so $C_d = O(c^d d!)$ for a constant c indpendent on the dimension. □

When d is a constant, it is correct to say that the algorithm runs in linear time. Still, that would be quite misleading. The constant factor C_d grows so fast as a function of d that this algorithm is useful only for rather small dimensions.

4.7* Smallest Enclosing Discs

The simple randomized technique we used above turns out to be surprisingly powerful. It can be applied not only to linear programming but to a variety of other optimization problems as well. In this section we shall look at one such problem.

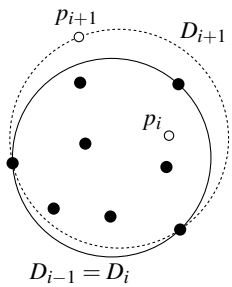


Consider a robot arm whose base is fixed to the work floor. The arm has to pick up items at various points and place them at other points. What would be a good position for the base of the arm? This would be somewhere "in the middle" of the points it must be able to reach. More precisely, a good position is at the center of the smallest disc that encloses all the points. This point minimizes the maximum distance between the base of the arm and any point it has to reach. We arrive at the following problem: given a set P of n points in the plane (the points on the work floor that the arm must be able to reach), find the *smallest enclosing disc* for P, that is, the smallest disc that contains all the points of P. This smallest enclosing disc is unique—see Lemma 4.14(i) below, which is a generalization of this statement.

As in the previous sections, we will give a randomized incremental algorithm for the problem: First we generate a random permutation $p_1, \ldots, p_n$ of the points in P. Let $P_i := \{p_1, \ldots, p_i\}$. We add the points one by one while we maintain D_i, the smallest enclosing disc of P_i.

In the case of linear programming, there was a nice fact that helped us to maintain the optimal vertex: when the current optimal vertex is contained in the next half-plane then it does not change, and otherwise the new optimal vertex lies on the boundary of the half-plane. Is a similar statement true for smallest enclosing discs? The answer is yes:

Lemma 4.13 *Let $2 < i < n$, and let P_i and D_i be defined as above. Then we have*
(i) If $p_i \in D_{i-1}$, then $D_i = D_{i-1}$.
(ii) If $p_i \notin D_{i-1}$, then p_i lies on the boundary of D_i.



We shall prove this lemma later, after we have seen how we can use it to design a randomized incremental algorithm that is quite similar to the linear programming algorithm.

Algorithm MINIDISC(P)
Input. A set P of n points in the plane.
Output. The smallest enclosing disc for P.
1. Compute a random permutation $p_1, \ldots, p_n$ of P.
2. Let D_2 be the smallest enclosing disc for $\{p_1, p_2\}$.
3. **for** $i \leftarrow 3$ **to** n
4. **do if** $p_i \in D_{i-1}$
5. **then** $D_i \leftarrow D_{i-1}$
6. **else** $D_i \leftarrow$ MINIDISCWITHPOINT($\{p_1, \ldots, p_{i-1}\}, p_i$)
7. **return** D_n

The critical step occurs when $p_i \notin D_{i-1}$. We need a subroutine that finds the smallest disc enclosing P_i, using the knowledge that p_i must lie on the boundary of that disc. How do we implement this routine? Let $q := p_i$. We use the same framework once more: we add the points of P_{i-1} in random order, and maintain the smallest enclosing disc of $P_{i-1} \cup \{q\}$ under the extra constraint that it should have q on its boundary. The addition of a point p_j will be facilitated by the following fact: when p_j is contained in the currently smallest enclosing disc then this disc remains the same, and otherwise it must have p_j on its boundary. So in the latter case, the disc has both q and p_j and its boundary. We get the following subroutine.

MINIDISCWITHPOINT(P, q)
Input. A set P of n points in the plane, and a point q such that there exists an enclosing disc for P with q on its boundary.
Output. The smallest enclosing disc for P with q on its boundary.
1. Compute a random permutation $p_1, \ldots, p_n$ of P.
2. Let D_1 be the smallest disc with q and p_1 on its boundary.
3. **for** $j \leftarrow 2$ **to** n
4. **do if** $p_j \in D_{j-1}$
5. **then** $D_j \leftarrow D_{j-1}$
6. **else** $D_j \leftarrow$ MINIDISCWITH2POINTS($\{p_1, \ldots, p_{j-1}\}, p_j, q$)
7. **return** D_n

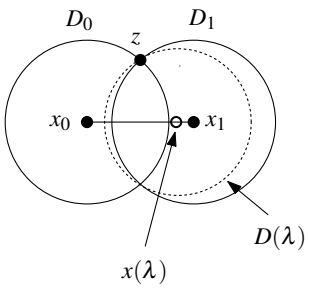
How do we find the smallest enclosing disc for a set under the restriction that two given points q_1 and q_2 are on its boundary? We simply apply the same approach one more time. Thus we add the points in random order and maintain the optimal disc; when the point p_k we add is inside the current disc we don't have to do anything, and when p_k is not inside the current disc it must be on the boundary of the new disc. In the latter case we have three points on the disc boundary: q_1, q_2, and p_k. This means there is only one disc left: the unique disc with q_1, q_2, and p_k on its boundary. This following routine describes this in more detail.

MINIDISCWITH2POINTS(P, q_1, q_2)
Input. A set P of n points in the plane, and two points q_1 and q_2 such that there exists an enclosing disc for P with q_1 and q_2 on its boundary.
Output. The smallest enclosing disc for P with q_1 and q_2 on its boundary.
1. Let D_0 be the smallest disc with q_1 and q_2 on its boundary.
2. **for** $k \leftarrow 1$ **to** n
3. **do if** $p_k \in D_{k-1}$
4. **then** $D_k \leftarrow D_{k-1}$
5. **else** $D_k \leftarrow$the disc with q_1, q_2, and p_k on its boundary
6. **return** D_n

This finally completes the algorithm for computing the smallest enclosing disc of a set of points. Before we analyze it, we must validate its correctness by proving some facts that we used in the algorithms. For instance, we used the

fact that when we added a new point and this point was outside the current optimal disc, then the new optimal disc must have this point on its boundary.

Lemma 4.14 *Let P be a set of points in the plane, let R be a possibly empty set of points with $P \cap R = \emptyset$, and let $p \in P$. Then the following holds:*
(i) If there is a disc that encloses P and has all points of R on its boundary, then the smallest such disc is unique. We denote it by $md(P,R)$.
(ii) If $p \in md(P \setminus \{p\}, R)$, then $md(P,R) = md(P \setminus \{p\}, R)$.
(iii) If $p \notin md(P \setminus \{p\}, R)$, then $md(P,R) = md(P \setminus \{p\}, R \cup \{p\})$.



Proof. (i) Assume that there are two distinct enclosing discs D_0 and D_1 with centers x_0 and x_1, respectively, and with the same radius. Clearly, all points of P must lie in the intersection $D_0 \cap D_1$. We define a continuous family $\{D(\lambda) \mid 0 \leqslant \lambda \leqslant 1\}$ of discs as follows. Let z be an intersection point of ∂D_0 and ∂D_1, the boundaries of D_0 and D_1. The center of $D(\lambda)$ is the point $x(\lambda) := (1-\lambda)x_0 + \lambda x_1$, and the radius of $D(\lambda)$ is $r(\lambda) := d(x(\lambda), z)$. We have $D_0 \cap D_1 \subset D(\lambda)$ for all λ with $0 \leqslant \lambda \leqslant 1$ and, in particular, for $\lambda = 1/2$. Hence, since both D_0 and D_1 enclose all points of P, so must $D(1/2)$. Moreover, $\partial D(1/2)$ passes through the intersection points of ∂D_0 and ∂D_1. Because $R \subset \partial D_0 \cap \partial D_1$, this implies that $R \subset \partial D(1/2)$. In other words, $D(1/2)$ is an enclosing disc for P with R on its boundary. But the radius of $D(1/2)$ is strictly less than the radii of D_0 and D_1. So whenever there are two distinct enclosing discs of the same radius with R on their boundary, then there is a smaller enclosing disc with R on its boundary. Hence, the smallest enclosing disc $md(P,R)$ is unique.

(ii) Let $D := md(P \setminus \{p\}, R)$. If $p \in D$, then D contains P and has R on its boundary. There cannot be any smaller disc containing P with R on its boundary, because such a disc would also be a containing disc for $P \setminus \{p\}$ with R on its boundary, contradicting the definition of D. It follows that $D = md(P,R)$.

(iii) Let $D_0 := md(P \setminus \{p\}, R)$ and let $D_1 := md(P,R)$. Consider the family $D(\lambda)$ of discs defined above. Note that $D(0) = D_0$ and $D(1) = D_1$, so the family defines a continous deformation of D_0 to D_1. By assumption we have $p \notin D_0$. We also have $p \in D_1$, so by continuity there must be some $0 < \lambda^* \leqslant 1$ such that p lies on the boundary of $D(\lambda^*)$. As in the proof of (i), we have $P \subset D(\lambda^*)$ and $R \subset \partial D(\lambda^*)$. Since the radius of any $D(\lambda)$ with $0 < \lambda < 1$ is strictly less than the radius of D_1, and D_1 is by definition the smallest enclosing disc for P, we must have $\lambda^* = 1$. In other words, D_1 has p on its boundary. □

Lemma 4.14 implies that MINIDISC correctly computes the smallest enclosing disc of a set of points. The analysis of the running time is given in the proof of the following theorem.

Theorem 4.15 *The smallest enclosing disc for a set of n points in the plane can be computed in $O(n)$ expected time using worst-case linear storage.*

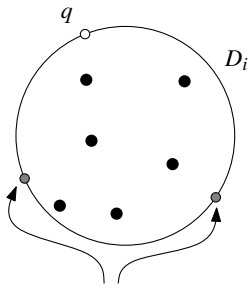
Proof. MINIDISCWITH2POINTS runs in $O(n)$ time because every iteration of the loop takes constant time, and it uses linear storage. MINIDISCWITHPOINT

and MINIDISC also need linear storage, so what remains is to analyze their expected running time.

The running time of MINIDISCWITHPOINT is $O(n)$ as long as we don't count the time spent in calls to MINIDISCWITH2POINTS. What is the probability of having to make such a call? Again we use backwards analysis to bound this probability: Fix a subset $\{p_1,\ldots,p_i\}$, and let D_i be the smallest disc enclosing $\{p_1,\ldots,p_i\}$ and having q on its boundary. Imagine that we remove one of the points $\{p_1,\ldots,p_i\}$. When does the smallest enclosing circle change? That happens only when we remove one of the three points on the boundary. One of the points on the boundary is q, so there are at most two points that cause the smallest enclosing circle to shrink. The probability that p_i is one of those points is $2/i$. (When there are more than three points on the boundary, then the probability that the smallest enclosing circle changes can only get smaller.) So we can bound the total expected running time of MINIDISCWITHPOINT by

$$O(n)+\sum_{i=2}^{n} O(i)\frac{2}{i} = O(n).$$

Applying the same argument once more, we find that the expected running time of MINIDISC is $O(n)$ as well. □



points that together with q define D_i

Algorithm MINIDISC can be improved in various ways. First of all, it is not necessary to use a fresh random permutation in every instance of subroutine MINIDISCWITHPOINT. Instead, one can compute a permutation once, at the start of MINIDISC, and pass the permutation to MINIDISCWITHPOINT. Furthermore, instead of writing three different routines, one could write a single algorithm MINIDISCWITHPOINTS(P,R) that computes $md(P,R)$ as defined in Lemma 4.14.

4.8 Notes and Comments

In this chapter we have studied an algorithmic problem that arises when one wants to manufacture an object using casting. Other manufacturing processes lead to challenging algorithmic problems as well, and a number of such problems have been studied within computational geometry over the past years—see for example the book by Dutta et al. [152] or the surveys by Janardan and Woo [220] and Bose and Toussaint [72].

The computation of the common intersection of half-planes is an old and well-studied problem. As we will explain in Chapter 11, the problem is dual to the computation of the convex hull of points in the plane. Both problems have a long history in the field, and Preparata and Shamos [323] already list a number of solutions. More information on the computation of 2-dimensional convex hulls can be found in the notes and comments of Chapter 1.

Computing the common intersection of half-spaces, which can be done in $O(n\log n)$ time in the plane and in 3-dimensional space, becomes a more computationally demanding problem when the dimension increases. The reason

is that the number of (lower-dimensional) faces of the convex polytope formed as the common intersection can be as large as $\Theta(n^{\lfloor d/2 \rfloor})$ [158]. So if the only goal is to find a feasible point, computing the common intersection explicitly soon becomes an unattractive approach.

Linear programming is one of the basic problems in numerical analysis and combinatorial optimization. It goes beyond the scope of this chapter to survey this literature, and we restrict ourselves to mentioning the simplex algorithm and its variants [139], and the polynomial-time solutions of Khachiyan [234] and Karmarkar [227]. More information on linear programming can be found in books by Chvátal [129] and Schrijver [339].

Linear programming as a problem in computational geometry was first considered by Megiddo [273], who showed that the problem of testing whether the intersection of half-spaces is empty is strictly simpler than the computation of the intersection. He gave the first deterministic algorithm for linear programming whose running time is of the form $O(C_d n)$, where C_d is a factor depending on the dimension only. His algorithm is linear in n for any fixed dimension. The factor C_d in his algorithm is 2^{2^d}. This was later improved to 3^{d^2} [130, 153]. More recently, a number of simpler and more practical randomized algorithms have been given [132, 346, 354]. There are a number of randomized algorithms whose running time is subexponential, but still not polynomial in the dimension [222, 267]. Finding a strongly polynomial algorithm, that is of combinatorial polynomial complexity, for linear programming is one of the major open problems in the area.

The simple randomized incremental algorithm for two and higher dimensions given here is due to Seidel [346]. Unlike in our presentation, he deals with unbounded linear programs by treating the parameter M symbolically. This is probably more elegant and efficient than the algorithm we present, which was chosen to demonstrate the relationship between unbounded d-dimensional linear programs and feasible $(d-1)$-dimensional ones. In Seidel's version, the factor C_d can be shown to be $O(d!)$.

The generalization to the computation of smallest enclosing discs is due to Welzl [385], who also showed how to find the smallest enclosing ball of a set of points in higher dimensions, and the smallest enclosing ellipse or ellipsoid. Sharir and Welzl further generalized the technique and introduced the notion of *LP-type problems*, which can be solved efficiently with an algorithm similar to the ones described here [189, 354]. Generally speaking, the technique is applicable to optimization problems where the solution either does not change when a new constraint is added, or the solution is partially defined by the new constraint so that the dimension of the problem is reduced. It has also been shown that the special properties of LP-type problems give rise to so-called Helly-type theorems [16].

Randomization is a technique that often produces algorithms that are simple and efficient. We will see more examples in the following chapters. The price we pay is that the running time is only an expected bound and—as we observed—there is a certain chance that the algorithm takes much longer. Some people take this as a reason to say that randomized algorithms cannot be trusted and

shouldn't be used (think of a computer in an intensive care station in a hospital, or in a nuclear power plant).

On the other hand, deterministic algorithms are only perfect in theory. In practice, any non-trivial algorithm may contain bugs, and even if we neglect this, there is the risk of hardware malfunction or "soft errors": single bits in core memory flipping under the influence of ambient α-radiation. Because randomized algorithms are often much simpler and have shorter code, the probability of such a mishap is smaller. Therefore the total probability that a randomized algorithm fails to produce the correct answer in time need not be larger than the probability that a deterministic algorithm fails. Moreover, we can always reduce the probability that the actual running time of a randomized algorithm exceeds its expected running time by allowing a larger constant in the expected running time.

4.9 Exercises

4.1 In this chapter we studied the casting problem for molds of one piece. A sphere cannot be manufactured in this manner, but it can be manufactured if we use a two-piece mold. Give an example of an object that cannot be manufactured with a two-piece mold, but that can be manufactured with a three-piece mold.

4.2 Consider the casting problem in the plane: we are given polygon $\mathcal{P}$ and a 2-dimensional mold for it. Describe a linear time algorithm that decides whether $\mathcal{P}$ can be removed from the mold by a single translation.

4.3 Suppose that, in the 3-dimensional casting problem, we do not want the object to slide along a facet of the mold when we remove it. How does this affect the geometric problem (computing a point in the intersection of half-planes) that we derived?

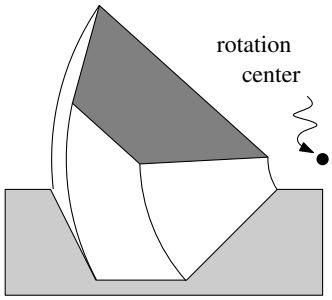
4.4 Let $\mathcal{P}$ be a castable simple polyhedron with top facet f. Let $\vec{d}$ be a removal direction for $\mathcal{P}$. Show that any line with direction $\vec{d}$ intersects $\mathcal{P}$ if and only if it intersects f. Also show that for any line ℓ with direction $\vec{d}$, the intersection $\ell \cap \mathcal{P}$ is connected.

4.5 Let $\mathcal{P}$ be a simple polyhedron with n vertices. If $\mathcal{P}$ is castable with some facet f as top facet, then a necessary condition is that the facets adjacent to f must lie completely to one side of h_f, the plane through f. (The reverse is not necessarily true, of course: if all adjacent facets lie to one side of h_f then $\mathcal{P}$ is not necessarily castable with f as top facet.) Give a linear time algorithm to compute all facets of $\mathcal{P}$ for which this condition holds.

4.6* Consider the restricted version of the casting problem in which we insist that the object is removed from its mold using a vertical translation (perpendicular to the top facet).

a. Prove that in this case there is always only a constant number of possible top facets.
b. Give a linear time algorithm that determines whether for a given object a mold exists under this restricted model.

4.7 Instead of removing the object from the mold by a single translation, we can also try to remove it by a single rotation. For simplicity, let's consider the planar variant of this version of the casting problem, and let's only look at clockwise rotations.



a. Give an example of a simple polygon $\mathcal{P}$ with top facet f that is not castable when we require that $\mathcal{P}$ should be removed from the mold by a single translation, but that is castable using rotation around a point. Also give an example of a simple polygon $\mathcal{P}$ with top facet f that is not castable when we require that $\mathcal{P}$ should be removed from the mold by a rotation, but that is castable using a single translation.
b. Show that the problem of finding a center of rotation that allows us to remove $\mathcal{P}$ with a single rotation from its mold can be reduced to the problem of finding a point in the common intersection of a set of half-planes.

4.8 The plane $z = 1$ can be used to represent all directions of vectors in 3-dimensional space that have a positive z-value. How can we represent all directions of vectors in 3-dimensional space that have a non-negative z-value? And how can we represent the directions of all vectors in 3-dimensional space?

4.9 Suppose we want to find *all* optimal solutions to a 3-dimensional linear program with n constraints. Argue that $\Omega(n \log n)$ is a lower bound for the worst-case time complexity of any algorithm solving this problem.

4.10 Let H be a set of at least three half-planes with a non-empty intersection such that not all bounding lines are parallel. We call a half-plane $h \in H$ *redundant* if it does not contribute an edge to $\bigcap H$. Prove that for any redundant half-plane $h \in H$ there are two half-planes $h', h'' \in H$ such that $h' \cap h'' \subset h$. Give an $O(n \log n)$ time algorithm to compute all redundant half-planes.

4.11 Give an example of a 2-dimensional linear program that is bounded, but where there is no lexicographically smallest solution.

4.12 Prove that RANDOMPERMUTATION(A) is correct, that is, prove that every possible permutation of A is equally likely to be the output. Also show that the algorithm is no longer correct (it no longer produces every permutation with equal probability) if we change the k in line 2 to n.

4.13 In the text we gave a linear time algorithm for computing a random permutation. The algorithm needed a random number generator that can produce a random integer between 1 and n in constant time. Now assume we have a restricted random number generator available that can only

generate a random bit (0 or 1) in constant time. How can we generate a random permutation with this restricted random number generator? What is the running time of your procedure?

4.14 Here is a paranoid algorithm to compute the maximum of a set A of n real numbers:

Algorithm PARANOIDMAXIMUM(A)
1. **if** card$(A) = 1$
2. **then return** the unique element $x \in A$
3. **else** Pick a random element x from A.
4. $x' \leftarrow$PARANOIDMAXIMUM$(A \setminus \{x\})$
5. **if** $x \leqslant x'$
6. **then return** x'
7. **else** Now we suspect that x is the maximum, but to be absolutely sure, we compare x with all card$(A) - 1$ other elements of A.
8. **return** x

What is the worst-case running time of this algorithm? What is the expected running time (with respect to the random choice in line 3)?

4.15 A simple polygon $\mathcal{P}$ is called *star-shaped* if it contains a point q such that for any point p in $\mathcal{P}$ the line segment $\overline{pq}$ is contained in $\mathcal{P}$. Give an algorithm whose expected running time is linear to decide whether a simple polygon is star-shaped.

4.16 On n parallel railway tracks n trains are going with constant speeds v_1, $v_2, \ldots, v_n$. At time $t = 0$ the trains are at positions $k_1, k_2, \ldots, k_n$. Give an $O(n \log n)$ algorithm that detects all trains that at some moment in time are leading. To this end, use the algorithm for computing the intersection of half-planes.

4.17* Show how to implement MINIDISC using a single routine MINIDISC-WITHPOINTS(P,R) that computes $md(P,R)$ as defined in Lemma 4.14. Your algorithm should compute only a single random permutation during the whole computation.