Chapter 20

Linear Programming in Low Dimensions

By Sariel Har-Peled, March 22, 2019[1]

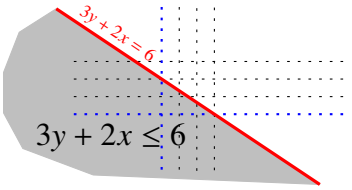
> At the sight of the still intact city, he remembered his great international precursors and set the whole place on fire with his artillery in order that those who came after him might work off their excess energies in rebuilding.
>
> The tin drum, Gunter Grass

In this chapter, we shortly describe (and analyze) a simple randomized algorithm for linear programming in low dimensions. Next, we show how to extend this algorithm to solve linear programming with violations. We then show how one can efficiently approximate the number constraints that one needs to violate to make a linear program feasible. This serves as a fruitful ground to demonstrate some of the techniques we visited already. Finally, we present an abstraction of linear programming, known as violator spaces, where there is no target function.

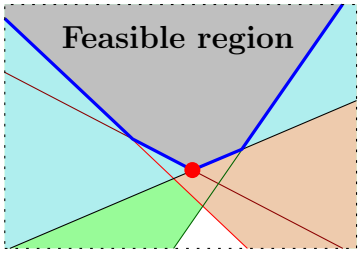
Our discussion is going to be somewhat intuitive. We will fill in the details and prove the correctness of our algorithms formally in the next chapter.

20.1. Linear programming

Assume we are given a set of n linear inequalities of the form $a_1x_1 + \cdots + a_dx_d \leq b$, where $a_1, \ldots, a_d, b$ are constants and $x_1, \ldots, x_d$ are the variables. In the ***linear programming*** (LP) problem, one has to find a ***feasible solution***, that is, a point $(x_1, \ldots, x_d)$ for which all the linear inequalities hold. In the following, we use the shorthand ***LPI*** to stand for ***linear programming instance***. Usually we would like to find a feasible point that maximizes a linear expression (referred to as the ***target function*** of the given LPI) of the form $c_1x_1 + \cdots + c_dx_d$, where $c_1, \ldots, c_d$ are prespecified constants.



The set of points complying with a linear inequality $a_1x_1 + \cdots + a_dx_d \leq b$ is a halfspace of $\mathbb{R}^d$ having the hyperplane $a_1x_1 + \cdots + a_dx_d = b$ as a boundary; see the figure on the right. As such, the feasible region of a LPI is the intersection of n halfspaces; that is, it is a ***polyhedron***. If the polyhedron is bounded, then it is a ***polytope***. The linear target function is no more than specifying a direction, such that we need to find the point inside the polyhedron which is extreme in this direction. If the polyhedron is unbounded in this direction, the optimal solution is ***unbounded***.



For the sake of simplicity of exposition, it will be easier to think of the direction for which one has to optimize as the negative x_d-axis direction. This can be easily realized by rotating the space such that the required direction is pointing downward. Since the feasible region is the intersection of convex sets (i.e., halfspaces), it is convex. As such, one can imagine the boundary of the feasible region as a vessel (with a convex interior). Next, we release a ball at the top of the vessel, and the ball rolls

down (by "gravity" in the direction of the negative x_{d}-axis) till it reaches the lowest point in the vessel and gets "stuck". This point is the optimal solution to the LPI that we are interested in computing.

In the following, we will assume that the given LPI is in general position. Namely, if we intersect k hyperplanes, induced by k inequalities in the given LPI (the hyperplanes are the result of taking each of this inequalities as an equality), then their intersection is a $(\mathsf{d}-k)$-dimensional affine subspace. In particular, the intersection of d of them is a point (referred to as a ***vertex***). Similarly, the intersection of any $\mathsf{d}+1$ of them is empty.

A polyhedron defined by an LPI with n constraints might have $O\left(n^{\lfloor \mathsf{d}/2 \rfloor}\right)$ vertices on its boundary (this is known as the upper-bound theorem [Grü03]). As we argue below, the optimal solution is a vertex. As such, a naive algorithm would enumerate all relevant vertices (this is a non-trivial undertaking) and return the best possible vertex. Surprisingly, in low dimension, one can do much better and get an algorithm with linear running time.

We are interested in the best vertex of the feasible region, while this polyhedron is defined implicitly as the intersection of halfspaces, and this hints to the quandary that we are in: We are looking for an optimal vertex in a large graph that is defined implicitly. Intuitively, this is why proving the correctness of the algorithms we present here is a non-trivial undertaking (as already mentioned, we will prove correctness in the next chapter).

20.1.1. A solution and how to verify it

Observe that an optimal solution of an LPI is either a vertex or unbounded. Indeed, if the optimal solution p lies in the middle of a segment s, such that s is feasible, then either one of its endpoints provides a better solution (i.e., one of them is lower in the x_{d}-direction than p) or both endpoints of s have the same target value. But then, we can move the solution to one of the endpoints of s. In particular, if the solution lies on a k-dimensional facet F of the boundary of the feasible polyhedron (i.e., formally F is a set with affine dimension k formed by the intersection of the boundary of the polyhedron with a hyperplane), we can move it so that it lies on a $(k-1)$-dimensional facet F' of the feasible polyhedron, using the proceeding argumentation. Using it repeatedly, one ends up in a vertex of the polyhedron or in an unbounded solution.

Thus, given an instance of LPI, the LP solver should output one of the following answers.

(A) **Finite.** The optimal solution is finite, and the solver would provide a vertex which realizes the optimal solution.

(B) **Unbounded.** The given LPI has an unbounded solution. In this case, the LP solver would output a ray ζ, such that the ζ lies inside the feasible region and it points down the negative x_{d}-axis direction.

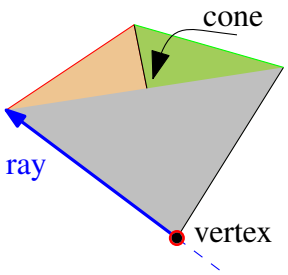
(C) **Infeasible.** The given LPI does not have any point which complies with all the given inequalities. In this case the solver would output $\mathsf{d}+1$ constraints which are infeasible on their own.

Lemma 20.1.1. *Given a set of d linear inequalities in $\mathbb{R}^{\mathsf{d}}$, one can compute the vertex induced by the intersection of their boundaries in $O(\mathsf{d}^3)$ time.*

Proof: Write down the system of equalities that the vertex must fulfill. It is a system of d equalities in d variables and it can be solved in $O(\mathsf{d}^3)$ time using Gaussian elimination. ■

A ***cone*** is the intersection of d constraints, where its apex is the vertex associated with this set of constraints. A set of such d constraints is a ***basis***. An intersection of $\mathsf{d}-1$ of the hyperplanes of a basis

forms a line and intersecting this line with the cone of the basis forms a ray. Clipping the same line to the feasible region would yield either a segment, referred to as an ***edge*** of the polyhedron, or a ray (if the feasible region is an unbounded polyhedron). An edge of the polyhedron connects two vertices of the polyhedron.



As such, one can think about the boundary of the feasible region as inducing a graph – its vertices and edges are the vertices and edges of the polyhedron, respectively. Since every vertex has d hyperplanes defining it (its basis) and an adjacent edge is defined by $\mathsf{d}-1$ of these hyperplanes, it follows that each vertex has $\binom{\mathsf{d}}{\mathsf{d}-1} = \mathsf{d}$ edges adjacent to it.

The following lemma tells us when we have an optimal vertex. While it is intuitively clear, its proof requires a systematic understanding of what the feasible region of a linear program looks like, and we delegate it to the next chapter.

Lemma 20.1.2. *Let L be a given LPI, and let $\mathcal{P}$ denote its feasible region. Let v be a vertex of $\mathcal{P}$, such that all the d rays emanating from v are in the upward x_{d}-axis direction (i.e., the direction vectors of all these d rays have positive x_{d}-coordinate). Then v is the lowest (in the x_{d}-axis direction) point in $\mathcal{P}$ and it is thus the optimal solution to L.*

Interestingly, when we are at a vertex v of the feasible region, it is easy to find the adjacent vertices. Indeed, compute the d rays emanating from v. For such a ray, intersect it with all the constraints of the LPI. The closest intersection point along this ray is the vertex u of the feasible region adjacent to v. Doing this naively takes $O\left(\mathsf{d}n + \mathsf{d}^{O(1)}\right)$ time.

Lemma 20.1.2 offers a simple algorithm for computing the optimal solution for an LPI. Start from a feasible vertex of the LPI. As long as this vertex has at least one ray that points downward, follow this ray to an adjacent vertex on the feasible polytope that is lower than the current vertex (i.e., compute the d rays emanating from the current vertex, and follow one of the rays that points downward, till you hit a new vertex). Repeat this till the current vertex has all rays pointing upward, by Lemma 20.1.2 this is the optimal solution. Up to tedious (and non-trivial) details this is the ***simplex*** algorithm.

We need the following lemma, whose proof is also delegated to the next chapter.

Lemma 20.1.3. *If L is an LPI in d dimensions which is not feasible, then there exist $\mathsf{d}+1$ inequalities in L which are infeasible on their own.*

Note that given a set of $\mathsf{d}+1$ inequalities, it is easy to verify (in polynomial time in d) if they are feasible or not. Indeed, compute the $\binom{\mathsf{d}+1}{\mathsf{d}}$ vertices formed by this set of constraints, and check whether any of these vertices are feasible (for these $\mathsf{d}+1$ constraints). If all of them are infeasible, then this set of constraints is infeasible.

20.2. Low-dimensional linear programming

20.2.1. An algorithm for a restricted case

There are a lot of tedious details that one has to take care of to make things work with linear programming. As such, we will first describe the algorithm for a special case and then provide the envelope required so that one can use it to solve the general case.

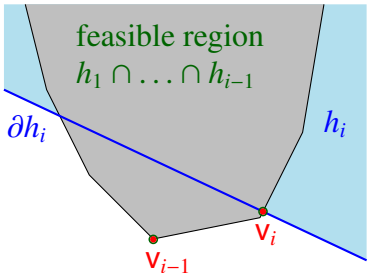
A vertex v is ***acceptable*** if all the d rays associated with it point upward (note that the vertex might not be feasible). The optimal solution (if it is finite) must be located at an acceptable vertex.

Input for the restricted case. The input for the restricted case is an LPI L, which is defined by a set of n linear inequalities in $\mathbb{R}^{\mathsf{d}}$, and a basis $B = \{h_1, \ldots, h_{\mathsf{d}}\}$ of an acceptable vertex.

Let $h_{\mathsf{d}+1}, \ldots, h_m$ be a random permutation of the remaining constraints of the LPI L.

We are looking for the lowest point in $\mathbb{R}^{\mathsf{d}}$ which is feasible for L. Our algorithm is randomized incremental. At the ith step, for $i > \mathsf{d}$, it will maintain the optimal solution for the first i constraints. As such, in the ith step, the algorithm checks whether the optimal solution v_{i-1} of the previous iteration is still feasible with the new constraint h_i (namely, the algorithm checks if v_{i-1} is inside the halfspace defined by h_i). If v_{i-1} is still feasible, then it is still the optimal solution, and we set $\mathsf{v}_i \leftarrow \mathsf{v}_{i-1}$.

The more interesting case is when $\mathsf{v}_{i-1} \notin h_i$. First, we check if the basis of v_{i-1} together with h_i forms a set of constraints which is infeasible. If so, the given LPI is infeasible, and we output $B(\mathsf{v}_{i-1}) \cup \{h_i\}$ as the proof of infeasibility.



Otherwise, the new optimal solution must lie on the hyperplane associated with h_i. As such, we recursively compute the lowest vertex in the $(\mathsf{d}-1)$-dimensional polyhedron $(\partial h_i) \cap \bigcap_{j=1}^{i-1} h_j$, where ∂h_i denotes the hyperplane which is the boundary of the halfspace h_i. This is a linear program involving $i-1$ constraints, and it involves $\mathsf{d}-1$ variables since the LPI lies on the $(\mathsf{d}-1)$-dimensional hyperplane ∂h_i. The solution found, v_i, is defined by a basis of $\mathsf{d}-1$ constraints in the $(\mathsf{d}-1)$-dimensional subspace ∂h_i, and adding h_i to it results in an acceptable vertex that is feasible in the original d-dimensional space. We continue to the next iteration.

Clearly, the vertex v_n is the required optimal solution.

20.2.1.1. Running time analysis

Every set of d constraints is feasible and computing the vertex formed by this constraint takes $O(\mathsf{d}^3)$ time, by Lemma 20.1.1.

Let X_i be an indicator variable that is 1 if and only if the vertex v_i is recomputed in the ith iteration (by performing a recursive call). This happens only if h_i is one of the d constraints in the basis of v_i. Since there are most d constraints that define the basis and there are at least $i - \mathsf{d}$ constraints that are being randomly ordered (as the first d slots are fixed), we have that the probability that $\mathsf{v}_i \neq \mathsf{v}_{i-1}$ is

$$\alpha_i = \mathbb{P}[X_i = 1] \leq \min\left(\frac{\mathsf{d}}{i-\mathsf{d}}, 1\right) \leq \frac{2\mathsf{d}}{i},$$

for $i \geq \mathsf{d}+1$, as can be easily verified.[2] So, let $T(m, \mathsf{d})$ be the expected time to solve an LPI with m constraints in d dimensions. We have that $T(\mathsf{d}, \mathsf{d}) = O(\mathsf{d}^3)$ by the above. Now, in every iteration, we need to check if the current solution lies inside the new constraint, which takes $O(\mathsf{d})$ time per iteration and $O(\mathsf{d}m)$ time overall.

Now, if $X_i = 1$, then we need to update each of the $i-1$ constraints to lie on the hyperplane h_i. The hyperplane h_i defines a linear equality, which we can use to eliminate one of the variables. This takes $O(\mathsf{d}i)$ time, and we have to do the recursive call. The probability that this happens is α_i. As such, we have

$$T(m, \mathsf{d}) = \mathbb{E}\left[O(m\mathsf{d}) + \sum_{i=\mathsf{d}+1}^{m} X_i(\mathsf{d}i + T(i-1, \mathsf{d}-1))\right]$$

[2] Indeed, $\frac{(\mathsf{d})+\mathsf{d}}{(i-\mathsf{d})+\mathsf{d}}$ lies between $\frac{\mathsf{d}}{i-\mathsf{d}}$ and $\frac{\mathsf{d}}{\mathsf{d}} = 1$.

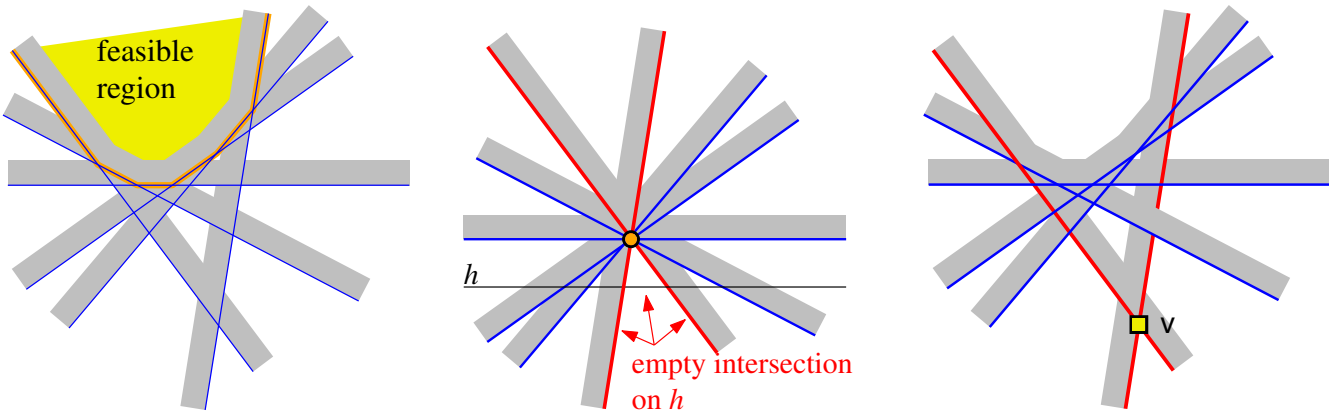


Figure 20.1: Demonstrating the algorithm for the general case: (a) given constraints and feasible region, (b) constraints moved to pass through the origin, and (c) the resulting acceptable vertex v.

$$
\begin{aligned}
&= O(m\mathsf{d}) + \sum_{i=\mathsf{d}+1}^{m} \alpha_i(\mathsf{d}i + T(i-1,\mathsf{d}-1)) \\
&= O(m\mathsf{d}) + \sum_{i=\mathsf{d}+1}^{m} \frac{2\mathsf{d}}{i}(\mathsf{d}i + T(i-1,\mathsf{d}-1)) \\
&= O\left(m\mathsf{d}^2\right) + \sum_{i=\mathsf{d}+1}^{m} \frac{2\mathsf{d}}{i} T(i-1,\mathsf{d}-1).
\end{aligned}
$$

Guessing that $T(m,\mathsf{d}) \leq c_{\mathsf{d}} m$, we have that

$$
T(m,\mathsf{d}) \leq \widehat{c}_1 m\mathsf{d}^2 + \sum_{i=\mathsf{d}+1}^{m} \frac{2\mathsf{d}}{i} c_{\mathsf{d}-1}(i-1) \leq \widehat{c}_1 m\mathsf{d}^2 + \sum_{i=\mathsf{d}+1}^{m} 2\mathsf{d}c_{\mathsf{d}-1} = \left(\widehat{c}_1\mathsf{d}^2 + 2\mathsf{d}c_{\mathsf{d}-1}\right)m,
$$

where $\widehat{c}_1$ is some absolute constant. We need that $\widehat{c}_1\mathsf{d}^2 + 2c_{\mathsf{d}-1}\mathsf{d} \leq c_{\mathsf{d}}$, which holds for $c_{\mathsf{d}} = O\left((3d)^{\mathsf{d}}\right)$ and $T(m,\mathsf{d}) = O\left((3d)^{\mathsf{d}} m\right)$.

Lemma 20.2.1. *Given an LPI with n constraints in d dimensions and an acceptable vertex for this LPI, then can compute the optimal solution in expected $O\left((3d)^{\mathsf{d}} n\right)$ time.*

20.2.2. The algorithm for the general case

Let L be the given LPI, and let L' be the instance formed by translating all the constraints so that they pass through the origin. Next, let h be the hyperplane $x_{\mathsf{d}} = -1$. Consider a solution to the LP L' when restricted to h. This is a $(\mathsf{d}-1)$-dimensional instance of linear programming, and it can be solved recursively.

If the recursive call on $L' \cap h$ returned no solution, then the d constraints that prove that the LP L' is infeasible on h corresponds to a basis in L of a vertex v which is acceptable in the original LPI. Indeed, as we move these d constraints to the origin, their intersection on h is empty (i.e., the "quadrant" that their intersection forms is unbounded only in the upward direction). As such, we can now apply the algorithm of Lemma 20.2.1 to solve the given LPI. See Figure 20.1.

If there is a solution to $L' \cap h$, then it is a vertex v on h which is feasible. Thus, consider the original set of $\mathsf{d}-1$ constraints in L that corresponds to the basis B of v. Let ℓ be the line formed by the intersection of the hyperplanes of B. It is now easy to verify that the intersection of the feasible region with this line is an unbounded ray, and the algorithm returns this unbounded (downward oriented) ray, as a proof that the LPI is unbounded.

Theorem 20.2.2. *Given an LP instance with n constraints defined over d variables, it can be solved in expected $O\big((3\mathsf{d})^{\mathsf{d}} n\big)$ time.*

Proof: The expected running time is

$$S(n,\mathsf{d}) = O(n\mathsf{d}) + S(n,\mathsf{d}-1) + T(m,\mathsf{d}),$$

where $T(m,\mathsf{d})$ is the time to solve an LP in the restricted case of Section 20.2.1. Indeed, we first solve the problem on the $(\mathsf{d}-1)$-dimensional subspace $h \equiv x_{\mathsf{d}} = -1$. This takes $O(\mathsf{d}n) + S(n,\mathsf{d}-1)$ time (we need to rewrite the constraints for the lower-dimensional instance, and that takes $O(\mathsf{d}n)$ time). If the solution on h is feasible, then the original LPI has an unbounded solution, and we return it. Otherwise, we obtained an acceptable vertex, and we can use the special case algorithm on the original LPI. Now, the solution to this recurrence is $O\big((3\mathsf{d})^{\mathsf{d}} n\big)$; see Lemma 20.2.1. ∎

20.3. Linear programming with violations

The problem. Let L be a linear program with d variables and n constraints, and let $k > 0$ be a parameter. We are interested in the optimal solution of L where we are allowed to throw away k constraints (i.e., the solution computed would ***violate*** these k constraints). A naive solution would be to try to throw away all possible subsets of k constraints, solve each one of these instances, and return the best solution found. This would require $O\big(n^{k+1}\big)$ time. Luckily, it turns out that one can do much better if the dimension is small enough.

Note that the given LPI might not be feasible but the solution violating k constraints might be unbounded. For the sake of simplicity of exposition, we will assume here that the optimal solution violating k constraints is bounded (i.e., it is a vertex).

Algorithm. We are given an LPI L with n constraints in d dimensions, a parameter k, and a confidence parameter $\varphi > 0$. The algorithm repeats the following $M = O(k^{\mathsf{d}} \log 1/\varphi)$ times:

(A) Pick, with a certain probability to be specified shortly, each constraint of L be in a random sample R.
(B) Compute the vertex u realizing the optimal solution for R if such a vertex exists.
(C) Count the number of constraints of L that u violates. If this number is at most k, the vertex u is a candidate for the solution.

Finally, the algorithm returns the best candidate vertex computed by step (C) above.

Analysis. The vertex realizing the optimal k-violated solution is a vertex v defined by d constraints (let B denote its basis) and is of depth k. We remind the reader that a point p has ***depth*** k (in L) if it is outside k halfspaces of L (namely, we complement each constraint of L, and p is contained inside k of these complemented hyperplanes). We denote the depth of p by $\mathtt{depth}(p)$. As such, we can use the depth estimation technique we encountered before. Specifically, if we pick each constraint of L to be in

a new instance of LP with probability $1/k$, then the probability that the new instance R would have all the elements of $\boldsymbol{B}$ in its random sample and would not contain any of the k constraints opposing v is

$$\alpha = \left(\frac{1}{k}\right)^{|B|}\left(1-\frac{1}{k}\right)^{\mathtt{depth}(\mathsf{v})} \geq \frac{1}{k^{\mathsf{d}}}\left(1-\frac{1}{k}\right)^{k} \geq \frac{1}{k^{\mathsf{d}}}\exp\left(-\frac{2}{k}k\right) \geq \frac{1}{8k^{\mathsf{d}}},$$

since $1-x \geq e^{-2x}$, for $0 < x < 1/2$. If this happens, then the optimal solution for R is v, since the basis of v is in R so the solution cannot be any lower and no constraint that violates v is in R.

Next, the algorithm amplified the probability of success by repeating this process $M = 8k^{\mathsf{d}}\ln(1/\varphi)$ times, returning the best solution found. The probability that in all these (independent) iterations we had failed to generate the optimal (violated) solution is at most

$$(1-\alpha)^{M} \leq \left(1-\frac{1}{8k^{\mathsf{d}}}\right)^{M} \leq \exp\left(-\frac{M}{8k^{\mathsf{d}}}\right) = \exp\left(-\ln\left(\frac{1}{\varphi}\right)\right) = \varphi.$$

Clearly, each iteration of the algorithm takes linear time.

Theorem 20.3.1. *Let L be a linear programming instance with n constraints over d variables, let $k > 0$ be a parameter, and let $\varphi > 0$ be a confidence parameter. Then one can compute the optimal solution to L violating at most k constraints of L, in $O\left(nk^{\mathsf{d}}\log(1/\varphi)\right)$ expected time. The solution returned is correct with probability $\geq 1-\varphi$.*

20.4. Approximate linear programming with violations

The magic of Theorem 20.3.1 is that it provides us with a linear programming solver which is robust and can handle a small number of factious constraints. But what happens if the number of violated constraints k is large?[3] As a concrete example, for $k = \sqrt{n}$ and an LPI with n constraints (defined over d variables) the algorithm for computing the optimal solution violating k constraints has running time roughly $O(n^{1+d/2})$. In this case, if one still wants a near linear running time, one can use random sampling to get an approximate solution in near linear time.

Lemma 20.4.1. *Let L be a linear program with n constraints over d variables, let $k > 0$ and $\varepsilon > 0$ be parameters. Then one can compute a solution to L violating at most $(1+\varepsilon)k$ constraints of L such that its value is at least as good as the optimal solution violating k constraints of L. The expected running time of the algorithm is*

$$O\left(n + n\min\left(\frac{\log^{\mathsf{d}+1} n}{\varepsilon^{2\mathsf{d}}}, \frac{\log^{\mathsf{d}+2} n}{k\,\varepsilon^{2\mathsf{d}+2}}\right)\right).$$

The algorithm succeeds with high probability.

Proof: Let $\rho = O\left(\frac{\mathsf{d}}{k\varepsilon^2}\ln n\right)$ and pick each constraint of L to be in L' with probability ρ. Next, the algorithm computes the optimal solution u in L' violating at most

$$k' = (1+\varepsilon/3)\rho k$$

constraints and returns this as the required solution.

[3] I am sure the reader guessed correctly the consequences of such a despicable scenario: The universe collapses and is replaced by a cucumber.

We need to prove the correctness of this algorithm. To this end, the reliable sampling lemma (L20.9.2) states that any vertex v of depth u in L has depth in the range

$$\left[(1-\varepsilon/3)u\rho,\, (1+\varepsilon/3)u\rho\right]$$

in L', and this holds with high probability, where $u \geq k$. Specifically, this holds with probability $\geq 1 - 1/n^{O(\mathsf{d})}$. We need this to hold for all the vertices in the original arrangement, and there are $\binom{n}{\mathsf{d}} = O(n^{\mathsf{d}})$ such vertices. Thus, this property holds for all the vertices with high probability.

In particular, let $\mathsf{v}_{\mathrm{opt}}$ be the optimal solution for L of depth k. With high probability, $\mathsf{v}_{\mathrm{opt}}$ has depth $\leq (1+\varepsilon/3)\rho k = k'$ in L', which implies that there is at least one vertex of depth $\leq k'$ in L'.

Now, we argue that any vertex of depth $\leq k'$ in L', with high probability, is a valid approximate solution. So, assume that we have a vertex v of depth β in L and its depth in L' is γ, where $\gamma \leq k'$.

By the reliable sampling lemma, we have that the depth γ of v in L' is in the range

$$\left[(1-\varepsilon/3)\beta\rho, (1+\varepsilon/3)\beta\rho\right].$$

This implies that $(1-\varepsilon/3)\beta\rho \leq k'$. Now, $k' = (1+\varepsilon/3)\rho k$, which implies that

$$(1-\varepsilon/3)\beta\rho \leq (1+\varepsilon/3)\rho k \quad \Longrightarrow \quad \beta \leq \frac{1+\varepsilon/3}{1-\varepsilon/3}k \leq (1+\varepsilon/3)(1+\varepsilon/2)k \leq (1+\varepsilon)k,$$

since $1/(1-\varepsilon/3) \leq 1+\varepsilon/2$ for $\varepsilon \leq 1$[④].

As for the running time, we are using the algorithm of Theorem 20.3.1, with $\varphi = 1/n^{O(\mathsf{d})}$. The input size is $O(\min(n, n\rho))$ and the depth threshold is k'. (The bound on the input size holds with high probability. We omit the easy but tedious proof of that using Chernoff's inequality.) As such, ignoring constants in the $O(\cdot)$ that depends exponentially on d, the running time is

$$\begin{aligned} O\Big(n + \min(n, n\rho)(k')^{\mathsf{d}} \log n\Big) &= O\Big(n + \min\big(n, n\rho\big)(\rho k)^{\mathsf{d}} \log n\Big) \\ &= O\left(n + n\min\left(\frac{\log^{\mathsf{d}+1} n}{\varepsilon^{2\mathsf{d}}},\ \frac{\log^{\mathsf{d}+2} n}{k\,\varepsilon^{2\mathsf{d}+2}}\right)\right). \end{aligned}$$

■

Note that the running time of Lemma 20.4.1 is linear if k is sufficiently large and ε is fixed.

20.5. LP-type problems

Interestingly, the algorithm presented for linear programming can be extended to more abstract settings. Indeed, assume we are given a set of constraints $\mathcal{H}$ and a function w, such that for any subset $G \subset \mathcal{H}$ it returns the value of the optimal solution of the constraint problem when restricted to G. We denote this value by $w(G)$. Our purpose is to compute $w(\mathcal{H})$.

For example, $\mathcal{H}$ is a set of points in $\mathbb{R}^{\mathsf{d}}$, and $w(G)$ is the radius of the smallest ball containing all the points of $G \subseteq \mathcal{H}$. As such, in this case, we would like to compute (the radius of) the smallest enclosing ball for $\mathcal{H}$.

We assume that the following axioms hold:

[④] Indeed $(1-\varepsilon/3)(1+\varepsilon/2) = 1 - \varepsilon/3 + \varepsilon/2 - \varepsilon^2/6 \geq 1$.

```
solveLPType(B_0, C)
    // B_0: initial basis, C: set of constraints
    ⟨c'_1, ..., c'_n⟩: random permutation of C \ B_0.
    for i = 1 to n do
        if compTarget(B_{i-1} ∪ {c'_i}) > compTarget(B_{i-1}) then
            T ← compBasis(B_{i-1} ∪ {c'_i})
            B_i ← solveLPType(T, B_0 ∪ {c'_1, ..., c'_i})
        else
            B_i ← B_{i-1}
    return B_n
```

Figure 20.2: The algorithm for solving LP-type problems.

1. (**Monotonicity**) For any $F \subseteq G \subseteq \mathcal{H}$, we have

$$w(F) \leq w(G).$$

2. (**Locality**) For any $F \subseteq G \subseteq \mathcal{H}$, with $-\infty < w(F) = w(G)$, and any $h \in \mathcal{H}$, if

$$w(G) < w(G \cup \{h\}), \quad \text{then} \quad w(F) < w(F \cup \{h\}).$$

If these two axioms hold, we refer to $(\mathcal{H}, w)$ as an ***LP-type*** problem. It is easy to verify that linear programming is an LP-type problem.

Definition 20.5.1. A ***basis*** is a subset $B \subseteq \mathcal{H}$ such that $w(B) > -\infty$ and $w(B') < w(B)$, for any proper subset B' of B.

As in linear programming, we have to assume that certain ***basic operations*** can be performed quickly:

(I) (**Violation test**) For a constraint h and a basis B, test whether h is violated by B or not. Namely, test if $w(B \cup \{h\}) > w(B)$. Let compTarget$(B \cup \{h\})$ be the provided function that returns $w(B \cup \{h\})$.

(II) (**Basis computation**) For a constraint h and a basis B, compute the basis of $B \cup \{h\}$. Let compBasis $(B \cup \{h\})$ be the procedure computing this basis.

We also need to assume (similar to the special case) that we are given an initial basis B_0 from which to start our computation. The ***combinatorial dimension*** of $(\mathcal{H}, w)$ is the maximum size of a basis of $\mathcal{H}$. A variant of the algorithm we presented for linear programming (the special case of Section 20.2.1) works in these settings. Indeed, start with B_0, and add the remaining constraints in a random order. At each step check if the new constraint violates the current solution, and if so, update the basis by performing a recursive call. Intuitively, the recursive call corresponds to solving a subproblem where some members of the basis are fixed. The algorithm is described in Figure 20.2.

20.5.1. Analysis of the algorithm

Lemma 20.5.2. *The algorithm solveLPType(B_0, C) terminates.*

Proof: Observe that every time that solveLPType calls recursively, it is done with a new initial basis T such that $w(T) > w(B_i) \geq w(B_0)$. Furthermore, the recursive call always returns a basis with value $\geq w(T)$. As such, the depth of the recursion is finite, and the algorithm terminates. ∎

Lemma 20.5.3. *In the end of the ith iteration of the loop of solveLPType(B_0, C), we have that*

$$w\left(B_0 \cup \left\{c'_1, \ldots c'_i\right\}\right) = w(B_i),$$

assuming that all calls to solveLPType with an initial basis T such that $w(T) > w(B_0)$ are successful.

Proof: The claim trivially holds for $i = 0$. So assume the claim holds for $i \leq k$ and we prove it for $i = k + 1$.

If $w\left(B_0 \cup \left\{c'_0, \ldots, c'_i\right\}\right) > w\left(B_0 \cup \left\{c'_0, \ldots, c'_{i-1}\right\}\right)$, then, by the locality axiom, we have $w\left(B_{i-1} \cup \left\{c'_i\right\}\right) > w(B_{i-1})$. This implies that c'_i violates B_{i-1}. The algorithm then recursively computes the basis T, which has $w(T) > w(B_{i-1}) \geq w(B_0)$. The algorithm then calls recursively on the set $B_0 \cup \left\{c'_0, \ldots, c'_i\right\}$, with an initial basis that has a higher value (i.e., there is progress being made before issuing this recursive call). By assumption, we have that the returned basis B_i is the desired basis of $B_0 \cup \left\{c'_0, \ldots, c'_i\right\}$.

If $w\left(B_{i-1} \cup c'_i\right) = w(B_{i-1})$, then, by locality, we have that

$$w\left(B_0 \cup \left\{c'_0, \ldots, c'_i\right\}\right) = w\left(B_0 \cup \left\{c'_0, \ldots, c'_{i-1}\right\}\right) = w(B_{i-1}) = w(B_i),$$

as $B_i = B_{i-1}$, which implies the claim. ∎

By arguing inductively on the value of the initial basis and using the above claim, we get the following.

Lemma 20.5.4. *The function solveLPType(B_0, C) returns a basis $B \subseteq B_0 \cup C$ such that $w(B) = w(B_0 \cup C)$.*

Lemma 20.5.5. *If the combinatorial dimension of $(\mathcal{H}, w)$ is d, then depth of the recursion of solveLPType(B_0, C) is bounded by d.*

Proof: To bound the depth of the recursion, we argue about the decreasing dimension of the recursive calls being made. Observe that if a constraint c'_i violates the current basis B_{i-1}, then any subset $X \subseteq B_0 \cup \left\{c'_1, \ldots c'_i\right\}$ that has $w(X) > w(B_{i-1})$ must have that $c'_i \in X$. Indeed, otherwise $X \subseteq Y_{i-1} = B_0 \cup \left\{c'_1, \ldots c'_{i-1}\right\}$ and then $w(X) \leq w(Y_{i-1}) = w(B_{i-1})$. As such, if a recursive call is of depth k, then k elements of the basis returned by this recursive call are fixed in advance; that is, there are k constraints in the initial basis (provided to this recursive call) that must appear in any basis output by this call. In particular, since no basis has size larger than d, it follows that the depth of the recursion is at most d. ∎

Theorem 20.5.6. *Let $(\mathcal{H}, w)$ be an instance of an LP-type problem with n constraints and with combinatorial dimension d. Assuming that the basic operations take constant time, we have that $(\mathcal{H}, w)$ can be solved by solveLPType using $\mathsf{d}^{O(\mathsf{d})} n$ basic operations (in expectation).*

Proof: If the number of constraints $n = O(\mathsf{d})$, then, by Lemma 20.5.5, the depth of the recursion is at most d, and the running time is $\mathsf{d}^{O(\mathsf{d})}$.

The analysis is now similar to the linear programming case. Indeed, the probability that in the ith iteration we need to perform a recursive call is bounded by $\min(1, \mathsf{d}/i)$. As such, if $T(n, i)$ is the expected running time, when the depth of the recursion is i and there are n constraints, then we have the recurrence

$$T(n, i) = O(n) + \sum_{k=1}^{n} \min\left(1, \frac{\mathsf{d}}{k}\right) T(k, i+1).$$

It is now easy to verify that $T(n, 0) = \mathsf{d}^{O(\mathsf{d})} n$, as the depth of recursion is bounded by d. ∎

20.5.2. Examples for LP-type problems

Smallest enclosing ball. Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, let $r(P)$ denote the radius of the smallest enclosing ball in $\mathbb{R}^{\mathsf{d}}$. Under general position assumptions, there are at most $\mathsf{d}+1$ points on the boundary of this smallest enclosing ball. We claim that this problem is an LP-type problem. Indeed, the basis in this case is the set of points determining the smallest enclosing ball. The combinatorial dimension is thus $\mathsf{d}+1$. The monotonicity property holds trivially. As for the locality property, assume that we have a set $Q \subseteq P$ such that $r(Q) = r(P)$. As such, P and Q have the same enclosing ball. Now, if we add a point p to Q and the radius of its minimum enclosing ball increases, then the ball enclosing P must also change (and get bigger) when we insert p into P. Thus, this is a LP-type problem, and it can be solved in linear time.

Theorem 20.5.7. *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, one can compute its smallest enclosing ball in (expected) linear time.*

Computing time of first intersection. Let $C(t)$ be a parameterized convex shape in $\mathbb{R}^{\mathsf{d}}$, such that $C(0)$ is empty and $C(t) \subsetneq C(t')$ if $t < t'$. We are given n such shapes $C_1, \ldots, C_n$, and we would like to decide the minimal t for which they all have a common intersection. Assume that given a point p and such a shape C, we can decide (in constant time) the minimum t for which $p \in C(t)$. Similarly, given (say) $\mathsf{d}+1$ of these shapes, we can decide, in constant time, the minimum t for which they intersect and this common point of intersection. We would like to find the minimum t for which they all intersect. Let us also assume that these shapes are well behaved in the sense that, for any t, we have $\lim_{\Delta \to 0} \mathrm{Vol}(C(t+\Delta) \setminus C(t)) = 0$ (namely, such a shape cannot "jump" – it grows continuously). It is easy to verify that this is an LP-type problem, and as such it can be solved in linear time.

Note that this problem is an extension of the previous problem. Indeed, if we place a ball of radius t at each point of P, then the problem of deciding the minimal t when all these growing balls have a non-empty intersection is equivalent to finding the minimum radius ball enclosing all points.

20.6. Violator spaces

A natural question is how to solve LP-type problems if there is no target function. It might not be obvious at first why such a scenario is interesting – we will address this issue shortly. Violator spaces is one way to extend LP-type problems to handle such problems. The basic idea is that every subset of constraints is mapped to a unique basis, every basis has size at most δ (δ is the dimension of the problem, and is conceptually a constant), and certain conditions on consistency and monotonicity hold. Computing the basis of a violator space is not as easy as solving LP-type problems, because without a clear notion of progress, one can cycle through bases (which is not possible for LP-type problems).

We review the formal definition of *violator spaces*, and then show that a variant of the algorithm we seen for LP-type problems works in this abstract setting.

20.6.1. Formal definition of violator space

Before delving into the abstract framework, let us consider the following concrete example – hopefully it would help the reader in keeping track of the abstraction.

Example 20.6.1. We have a set H of n segments in the plane, and we would like to compute the vertical trapezoid of $\mathcal{A}^{|}(H)$ that contains, say, the origin, where $\mathcal{A}^{|}(H)$ denote the vertical decomposition of the

arrangement formed by the segments of H. For a subset $X \subseteq H$, let $\square_X$ be the vertical trapezoid in $\mathcal{A}^{|}(X)$ that contains the origin. The vertical trapezoid $\square_X$ is defined by at most four segments, which are the *basis* of X. A segment $f \in H$ *violates* $\square = \square_X$, if it intersects the interior of $\square_X$. The set of segments of H that intersect the interior of $\square$, denoted by $\mathrm{cl}(\square)$ or $\mathrm{cl}(X)$, is the *conflict list* of $\square$.

Somewhat informally, a violator space identifies a vertical trapezoid $\square = \square_X$, by its conflict list $\mathrm{cl}(X)$, and not by its geometric realization (i.e., $\square$).

Definition 20.6.2. A ***violator space*** is a pair $\mathcal{V} = (H, \mathrm{cl})$, where H is a finite set of ***constraints***, and $\mathrm{cl} : 2^H \to 2^H$ is a function, such that:

- **Consistency**: For all $X \subseteq H$, we have that $\mathrm{cl}(X) \cap X = \emptyset$.
- **Locality**: For all $X \subseteq Y \subseteq H$, if $\mathrm{cl}(X) \cap Y = \emptyset$ then $\mathrm{cl}(X) = \mathrm{cl}(Y)$.
- **Monotonicity**: For all $X \subseteq Y \subseteq Z \subseteq H$, if $\mathrm{cl}(X) = \mathrm{cl}(Z)$ then $\mathrm{cl}(X) = \mathrm{cl}(Y) = \mathrm{cl}(Z)$.

A set $B \subseteq X \subseteq H$ is a ***basis*** of X, if $\mathrm{cl}(B) = \mathrm{cl}(X)$, and for any proper subset $B' \subset B$, we have that $\mathrm{cl}(B') \neq \mathrm{cl}(B)$. The ***combinatorial dimension***, denoted by δ, is the maximum size of a basis.

Note that consistency and locality implies monotonicity. For the sake of concreteness, it is also convenient to assume the following (this is strictly speaking not necessary for the algorithm).

Definition 20.6.3. For any $X \subseteq H$ there is a unique ***cell*** $\square_X$ associated with it, where for any $X, Y \subseteq H$, we have that if $\mathrm{cl}(X) \neq \mathrm{cl}(Y)$ then $\square_X \neq \square_Y$. Consider any $X \subseteq H$, and any $f \in H$. For $\square = \square_X$, the constraint f ***violates*** $\square$ if $f \in \mathrm{cl}(X)$ (or alternatively, f *violates* X).

Finally, we assume that the following two basic operations are available:

- violate(f, B): Given a basis B (or its cell $\square = \square_B$) and a constraint f, it returns true $\iff$ f violates $\square$.
- compBasis(X): Given a set X with at most $(\delta+1)^2$ constraints, this procedure computes basis(X), where δ is the combinatorial dimension of the violator space[5]. For δ a constant, we assume that this takes constant time.

20.6.1.1. Linear programming via violator spaces

Consider an instance I of linear programming in $\mathbb{R}^d$ – here the LP is defined by a collection of linear inequalities with d variables. The instance I induces a polytope $\mathcal{P}$ in $\mathbb{R}^d$, which is the *feasible domain* – specifically, every inequality induces a halfspace, and their intersection is the polytope.

The following interpretation of the feasible polytope is somewhat convoluted, but serves as a preparation for the next example. The vertices V of the polytope $\mathcal{P}$ induce a triangulation (assuming general position) of the sphere of directions, where a direction v belongs to a vertex p, if and only if p is an extreme vertex of $\mathcal{P}$ in the direction of v. Now, the objective function of I specifies a direction v_I, and in solving the LP, we are looking for the extreme vertex of $\mathcal{P}$ in this direction.

Put differently, every subset H of the constraints of I, defines a triangulation $\mathcal{T}(H)$ of the sphere of directions. So, let the cell of H, denoted by $\square = \square_H$, be the spherical triangle in this decomposition that contains v_I. The basis of H is the subset of constraints that define $\square_H$. A constraint f of the LP violates $\square$ if the vertex induced by the basis basis(H) (in the original space), is on the wrong side of f.

Thus solving the LP instance $I = (H, v_I)$ is no more than performing a point location query in the spherical triangulation $\mathcal{T}(H)$, for the spherical triangle that contains v_I.

[5] We consider basis(X) to be unique (that is, we assume implicitly that the input is in general position). This can be enforced by using lexicographical ordering, if necessary, among the defining bases always using the lexicographically minimum basis.

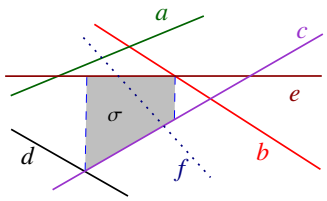


Figure 20.3: Vertical decomposition and the Clarkson-Shor framework. The defining set of σ is $D(\sigma) = \{a, b, c, d\}$, and its stopping set is $K(\sigma) = \{f\}$.

Remark. An alternative, and somewhat more standard, way to see this connection is via geometric duality [Har11, Chapter 25] (which is of course *not* LP duality). The duality maps the upper envelope of hyperplanes to a convex-hull in the dual space (i.e., we assume here that in the given LP all the halfspaces (i.e., each constraint corresponds to a halfspace) contain the positive ray of the x_d-axis – otherwise, we need to break the constraints of the LP into two separate families, and apply this reduction separately to each family). Then, extremal query on the feasible region of the LP, in the dual, becomes a vertical ray shooting query on the upper portion of the convex-hull of the dual points – that is, a point location query in the projection of the triangulation of the upper portion of the convex-hull of the dual points.

20.6.1.2. Point location via violator spaces

Example 20.6.1 hints to a more general setup. So consider a space decomposition into canonical cells induced by a set of objects. For example, segments in the plane, with the canonical cells being the vertical trapezoids. More generally, consider any decomposition of a domain into simple canonical cells induced by objects, which falls under the Clarkson-Shor framework [Cla88, CS89] (defined formally below). Examples of this include point location in a (i) Delaunay triangulation, (ii) bottom vertex triangulation in an arrangement of hyperplanes, (iii) and vertical decomposition of arrangements of curves and surfaces in two or three dimensions, to name a few.

In the following, we need the Clarkson-Shor technique.

Definition 20.6.4 (Clarkson-Shor framework [Cla88, CS89]). Let $\mathcal{D}$ be an underlying domain, and let H be a set of objects, such that any subset $R \subseteq H$ decomposes $\mathcal{D}$ into canonical cells $\mathcal{F}(R)$. This decomposition complies with the Clarkson-Shor framework, if we have that for every cell $\square$ that arises from such a decomposition has a *defining set* $D(\square) \subseteq H$. The size of such a defining set is assumed to be bounded by a constant δ. The *stopping set* (or conflict list) $K(\square)$ of $\square$ is the set of objects of H such that including any object of $K(\square)$ in R prevents $\square$ from appearing in $\mathcal{F}(R)$. We require that for any $R \subseteq H$, the following conditions hold:

(i) For any $\square \in \mathcal{F}(R)$, we have $D(\square) \subseteq R$ and $R \cap K(\square) = \emptyset$.
(ii) If $D(\square) \subseteq R$ and $K(\square) \cap R = \emptyset$, then $\square \in \mathcal{F}(R)$.

For a detailed discussion of the Clarkson-Shor framework, see Har-Peled [Har11, Chapter 8].

We next provide a quick example for the reader unfamiliar with this framework.

Example 20.6.5. Consider a set of segments H in the plane. For a subset $R \subseteq H$, let $\mathcal{A}^{|}(R)$ denote the ***vertical decomposition*** of the plane formed by the arrangement $\mathcal{A}(R)$ of the segments of R. This is the partition of the plane into interior disjoint vertical trapezoids formed by erecting vertical walls through each vertex of $\mathcal{A}(R)$. Here, each object is a segment, a region is a vertical trapezoid, and $\mathcal{F}(R)$ is the set of vertical trapezoids in $\mathcal{A}^{|}(R)$. Each trapezoid $\sigma \in \mathcal{F}(R)$ is defined by at most four segments (or lines) of R that define the region covered by the trapezoid σ, and this set of segments is $D(\sigma)$. Here, $K(\sigma)$ is the set of segments of H intersecting the interior of the trapezoid σ (see Figure 20.3).

```
solveVS(W, X):
    ⟨f_1, ..., f_m⟩: A random permutation of the constraints of X.
    B_0 ← compBasis(W)
    for i = 1 to m do
        if violate(f_i, B_{i-1}) then
            B_i ← solveVS(W ∪ B_{i-1} ∪ {f_i}, {f_1, ..., f_i})
        else
            B_i ← B_{i-1}
    return B_m
```

Figure 20.4: The algorithm for solving violator space problems. The parameter W is a set of $O(\delta^2)$ witness constraints, and X is a set of m constraints. The function return basis($W \cup X$). To solve a given violator space, defined implicitly by the set of constraints H, and the functions violate and compBasis, one calls solveVS($\{\}, H$).

Lemma 20.6.6. *Consider a canonical decomposition of a domain into simple cells, induced by a set of objects, that complies with the Clarkson-Shor framework [Cla88, CS89]. Then, performing a point location query in such a domain is equivalent to computing a basis of a violator space.*

Proof: This follows readily from definition, but we include the details for the sake of completeness. We use the notations of Definition 20.6.4 above.

So consider a fixed point $p \in \mathcal{D}$, and the task at hand is to compute the cell $\square \in \mathcal{F}(H)$ that contains p (here, assuming general position implies that there is a unique such cell). In particular, for $R \subseteq H$, we define $B = \text{basis}(R)$ to be the defining set of the cell $\square$ of $\mathcal{F}(R)$ that contains p, and the conflict list to be $\text{cl}(R) = K(\square)$.

We claim that computing basis(H) is a violator space problem. We need to verify the conditions of Definition 20.6.2, which is satisfyingly easy. Indeed, consistency is condition ((i)), and locality is condition ((ii)) in Definition 20.6.4. (We remind the reader that consistency and locality implies monotonicity, and thus we do not have to verify that it holds.)

It seems that for all of these point location problems, one can solve them directly using the LP-type technique. However, stating these problems as violator space instances is more natural as it avoids the need to explicitly define an artificial ordering over the bases, which can be quite tedious and not immediate.

20.6.2. The algorithm for computing the basis of a violator space

The input is a violator space $\mathcal{V} = (H, \text{cl})$ with $n = |H|$ constraints, having combinatorial dimension δ.

20.6.2.1. Description of the algorithm

The algorithm is a variant of Seidel's algorithm [Sei91] – it picks a random permutation of the constraints, and computes recursively in a randomized incremental fashion the basis of the solution for the first i constraints. Specifically, if the ith constraint violates the basis B_{i-1} computed for the first $i-1$ constraints, it calls recursively, adding the constraints of B_{i-1} and the ith constraint to the set of constraints that must be included whenever computing a basis (in the recursive calls). The resulting code is depicted in Figure 20.4.

The only difference with the original algorithm of Seidel, is that the recursive call gets the set $W \cup B_{i-1} \cup \{f_i\}$ instead of $\text{basis}(B_{i-1} \cup \{f_i\})$ (which is a smaller set). This modification is required because of the potential cycling between bases in a violator space.

20.6.2.2. Analysis

The key observation is that the depth of the recursion of solveVS is bounded by δ, where δ is the combinatorial dimension of the violator space. Indeed, if f_i violates a basis, the constraints added to the witness set W guarantee that any subsequent basis computed in the recursive call contains f_i, as testified by the following lemma.

Lemma 20.6.7. *Consider any set $X \subseteq H$. Let $B = \text{basis}(X)$, and let f be a constraint in $H \setminus X$ that violates B. Then, for any subset Y such that $B \cup \{f\} \subseteq Y \subseteq X \cup \{f\}$, we have that $f \in \text{basis}(Y)$.*

Proof: Assume that this is false, and let Y be the bad set with $B' = \text{basis}(Y)$, such that $f \notin B'$. Since $f \in Y$, by consistency, $f \notin \text{cl}(Y)$, see Definition 20.6.2. By definition $\text{cl}(Y) = \text{cl}(B')$, which implies that $f \notin \text{cl}(B')$; that is, f does not violate B'.

Now, by monotonicity, we have $\text{cl}(Y) = \text{cl}(Y \setminus \{f\}) = \text{cl}(B')$.

By assumption, $B \subseteq Y \setminus \{f\}$, which implies, again by monotonicity, as $B \subseteq Y \setminus \{f\} \subseteq X$, that $\text{cl}(X) = \text{cl}(Y \setminus \{f\}) = \text{cl}(B)$, as $B = \text{basis}(X)$. But that implies that $\text{cl}(B) = \text{cl}(Y \setminus \{f\}) = \text{cl}(B')$. As $f \notin \text{cl}(Y)$, this implies that f does not violate B, which is a contradiction. ∎

Lemma 20.6.8. *The depth of the recursion of solveVS, see Figure 20.4, is at most δ, where δ is the combinatorial dimension of the given instance.*

Proof: Consider a sequence of k recursive calls, with $W_0 \subseteq W_1 \subseteq W_2 \subseteq \cdots \subseteq W_k$ as the different values of the parameter W of solveVS, where $W_0 = \emptyset$ is the value in the top-level call. Let g_j, for $j = 1, \ldots, k$, be the constraint whose violation triggered the jth level call. Observe that $g_j \in W_j$, and as such all these constraints must be distinct (by consistency). Furthermore, we also included the basis B'_j, that g_j violates, in the witness set W_j. As such, we have that $W_j = \{g_1, \ldots, g_j\} \cup B'_1 \cup \cdots \cup B'_j$. By Lemma 20.6.7, in any basis computation done inside the recursive call $\mathit{solveVS}(W_j, \ldots)$, it must be that $g_j \in \text{basis}(W_t)$, for any $t \geq j$. As such, we have $g_1, \ldots, g_k \in \text{basis}(W_k)$. Since a basis can have at most δ elements, this is possible only if $k \leq \delta$, as claimed. ∎

Theorem 20.6.9. *Given an instance of violator space $\mathcal{V} = (H, \text{cl})$ with n constraints, and combinatorial dimension δ, the algorithm solveVS$(\emptyset, H)$, see Figure 20.4, computes* $\text{basis}(H)$. *The expected number of violation tests performed is bounded by $O(\delta^{\delta+1} n)$. Furthermore, the algorithm performs in expectation $O\left((\delta \ln n)^{\delta}\right)$ basis computations (on sets of constraints that contain at most $\delta(\delta+1)$ constraints).*

In particular, for constant combinatorial dimension δ, with violation test and basis computation that takes constant time, this algorithm runs in $O(n)$ expected time.

Proof: Lemma 20.6.8 implies that the recursion tree has bounded depth, and as such this algorithm must terminate. The correctness of the result follows by induction on the depth of the recursion. By Lemma 20.6.8, any call of depth δ, cannot find any violated constraint in its subproblem, which means that the returned basis is indeed the basis of the constraints specified in its subproblem. Now, consider a recursive call at depth $j < \delta$, which returns a basis B_m when called on the constraints $\langle f_1, \ldots, f_m \rangle$. The basis B_m was computed by a recursive call on some prefix $\langle f_1, \ldots, f_i \rangle$, which was correct by (reverse) induction on the depth, and none of the constraints $f_{i+1}, \ldots, f_m$ violates B_m, which implies that the returned basis is a basis for the given subproblem. Thus, the result returned by the algorithm is correct.

Let $C_k(m)$ be the expected number of basis computations performed by the algorithm when run at recursion depth k, with m constraints. We have that $C_\delta(m) = 1$, and $C_k(m) = 1 + \mathbb{E}\left[\sum_{i=1}^m X_i C_{k+1}(i)\right]$, where X_i is an indicator variable that is one, if and only if the insertion of the ith constraint caused a recursive call. We have, by backward analysis, that $\mathbb{E}[X_i] = \mathbb{P}[X_i = 1] \le \min(\delta/i, 1)$. As such, by linearity of expectations, we have $C_k(m) \le 1 + \sum_{i=1}^m \min\left(\frac{\delta}{i}, 1\right) C_{k+1}(i)$. As such, we have

$$C_{\delta-1}(m) \le 1 + \sum_{i=1}^{m} \min\left(\frac{\delta}{i}, 1\right) C_\delta(i) = \delta + \sum_{i=\delta}^{m} \frac{\delta}{i} \le \delta + \delta \ln(m/\delta) \le \delta \ln(m),$$

assuming $\delta \ge e$. We conclude that $C_{\delta-1}(m) \le \delta \ln(m)$ and $C_0(m) = O\left((\delta \ln m)^\delta\right)$.

As for the expected number of violation tests, a similar analysis shows that $V_\delta(m) = m$ and $V_{\delta-j}(m) = m + \sum_{i=1}^m \min\left(\frac{\delta}{i}, 1\right) V_{\delta-(j-1)}(i)$. As such, assuming inductively that $V_{\delta-(j-1)}(m) \le j\delta^{j-1}m$, we have that

$$V_{\delta-j}(m) \le m + \sum_{i=1}^{m} \frac{\delta}{i} j\delta^{j-1} i \le m + j\delta^j m \le (j+1)\delta^j m,$$

implying that $V_0(m) = O(\delta^{\delta+1}m)$, which also bounds the running time. ∎

Remark 20.6.10. While the constants in Theorem 20.6.9 are not pretty, we emphasize that the O notation in the bounds do not hide constants that depends on δ.

20.7. Bibliographical notes

History. Linear programming has a fascinating history. It can be traced back to the early 20th century. It started in earnest in 1939 when L. V. Kantorovich noticed the importance of certain types of linear programming problems for economic planning.

Dantzig, in 1947, invented the simplex method for solving LP problems for the US Air Force planning problems. T. C. Koopmans, in 1947, showed that LP provides the right model for the analysis of classical economic theories. In 1975, both Koopmans and Kantorovich got the Nobel prize for economics. Dantzig probably did not receive it because his work was too mathematical. So it goes. Interestingly, Kantorovich is the only Russian who was awarded the Nobel prize in economics during the cold war.

The simplex algorithm was developed before computers (and computer science) really existed in wide usage, and its standard description is via a careful maintenance of a tableau of the LP, which is easy to handle by hand (this might also explain the unfortunate name "linear programming"). As such, the usual description of the simplex algorithm is somewhat mysterious and counterintuitive (at least for the author). Furthermore, since the universe is not in general position (as we assumed), there are numerous technical difficulties (that we glossed over) in implementing any of these algorithms, and the descriptions of the simplex algorithm usually detail how to handle these cases. See the book by Vanderbei [Van97] for an accessible description of this topic.

Linear programming in low dimensions. The first to realize that linear programming can be solved in linear time in low dimensions was Megiddo [Meg83, Meg84]. His algorithm was deterministic but considerably more complicated than the randomized algorithm we present. Clarkson [Cla95] showed how to use randomization to get a simple algorithm for linear programming with running time $O(\mathsf{d}^2 n + noise)$, where the noise is a constant exponential in d. Our presentation follows the paper by Seidel [Sei91].

Surprisingly, one can achieve running time with the noise being subexponential in d. This follows by plugging the subexponential algorithms of Kalai [Kal92] or Matoušek et al. [MSW96] into Clarkson's algorithm [Cla95]. The resulting algorithm has expected running time $O\left(\mathsf{d}^2 n + \exp\left(c\sqrt{\mathsf{d}\log\mathsf{d}}\right)\right)$, for some constant c. See the survey by Goldwasser [Gol95] for more details.

More information on Clarkson's algorithm. Clarkson's algorithm contains some interesting new ideas. (The algorithm of Matoušek et al. [MSW96] is somewhat similar to the algorithm we presented.)

Observe that if the solution for a random sample R is being violated by a set X of constraints, then X must contain (at least) one constraint which is in the basis of the optimal solution. Thus, by picking R to be of size (roughly) $\sqrt{n}$, we know that it is a $1/\sqrt{n}$-net and there would be at most $\sqrt{n}$ constraints violating the solution of R; see Theorem 20.9.1$_{\text{p18}}$. Thus, repeating this d times, at each stage solving the problem on the collected constraints from the previous iteration, together with the current random sample, results in a set of $O\left(\mathsf{d}\sqrt{n}\right)$ constraints containing the optimal basis. Now solve recursively the linear program on this (greatly reduced) set of constraints. Namely, we spent $O\left(\mathsf{d}^2 n\right)$ time (d times checking if the n constraints violate a given solution), called recursively d times on "small" subproblems of size (roughly) $O\left(\sqrt{n}\right)$, resulting in a fast algorithm.

An alternative algorithm uses the same observation, by using the reweighting technique. Here each constraint is sampled according to its weight (which is initially 1). By doubling the weight of the violated constraints, one can argue that after a small number of iterations, the sample would contain the required basis, while being small.

Clarkson's final algorithm works by combining these two algorithms together.

Linear programming with violations. The algorithm of Section 20.3 seems to be new, although it is implicit in the work of Matoušek [Mat95], which presents a slightly faster deterministic algorithm. The first paper on this problem (in two dimensions) is due to Everett et al. [ERK96]. This was extended by Matoušek to higher dimensions [Mat95]. His algorithm relies on the idea of computing all $O(k^{\mathsf{d}})$ local maximas in the "k-level" explicitly, by traveling between them. This is done by solving linear programming instances which are "similar". As such, these results can be further improved using techniques for dynamic linear programming that allow insertion and deletions of constraints; see the work by Chan [Cha96]. Chan [Cha05] showed how to further improve these algorithms for dimensions 2, 3, and 4, although these improvements disappear if k is close to linear.

The idea of approximate linear programming with violations is due to Aronov and Har-Peled [AH08], and our presentation follows their results. Using more advanced data-structures, these results can be further improved (as far as the polylog noise is concerned); see the work by Afshani and Chan [AC09].

LP-type problems. The notion of LP-type algorithms is mentioned in the work of Sharir and Welzl [SW92]. They also showed that deciding if a set of (axis parallel) rectangles can be pierced by three points is an LP-type problem (quite surprising as the problem has no convex programming flavor). Our presentation is influenced by the subsequent work by Matoušek et al. [MSW96]. (But our analysis is significantly weaker, and it is inspired by the author's laziness.) Our example of computing the first intersection of growing convex sets is motivated by the work of Amenta [Ame94] on the connection between LP-type problems and Helly-type theorems.

Intuitively, any lower-dimensional convex programming problem is a natural candidate to be solved using LP-type techniques.

Violator spaces. The notion of *violator spaces* was introduced and studied in the following papers [Rüs07, Ško07, GMRŠ06, GMRŠ08, BG11]. See Šavroň [Ško07] for an example of cycling that is caused by not having an explicit target function. It is known that Clarkson's algorithm [Cla95] works for violator spaces [BG11]. Our exposition follows the work by the author [Har16], which showed that violator spaces can be solve in expected linear time using constant space.

20.8. Exercises

Exercise 20.1 (Blue/red separation). Let R and B be sets of red and blue points, respectively, in $\mathbb{R}^{\mathsf{d}}$. Describe a linear time algorithm that computes the maximum width slab (i.e., the region enclosed by two parallel hyperplanes) that separates R from B.

Exercise 20.2 (Approximate blue/red separation). Let R and B be sets of red and blue points, respectively, in the high-dimensional Euclidean space $\mathbb{R}^{\mathsf{d}}$. Let γ be the width of the maximum width slab separating R from B, and let $\varepsilon > 0$ be a parameter.

Present an algorithm, with running time $O\left(\mathsf{d}n\left(\frac{\operatorname{diam}(B \cup R)}{\gamma}\right)^{c}\right)$, that computes a slab separating R from B of width $\geq (1-\varepsilon)\gamma$, where c is some absolute constant independent of d.

(Hint: Maintain two points x and y that are inside the convex hulls of R and B, respectively. Consider the slab defined by hyperplanes perpendicular to the segment xy and that pass through the points x and y. At each step either this slab is "good enough" or, alternatively, it contains a point of $R \cup B$ deep inside the slab. Consider such a bad point, and update either x or y to define a smaller slab. Bound the speed of convergence of this process.)

20.9. From previous chapters

Theorem 20.9.1 (ε-net theorem, [HW87]). *Let $(X, \mathcal{R})$ be a range space of VC dimension δ, let x be a finite subset of X, and suppose that $0 < \varepsilon \leq 1$ and $\varphi < 1$. Let N be a set obtained by m random independent draws from x, where*

$$m \geq \max\left(\frac{4}{\varepsilon}\lg\frac{4}{\varphi}, \frac{8\delta}{\varepsilon}\lg\frac{16}{\varepsilon}\right). \tag{20.1}$$

Then N is an ε-net for x with probability at least $1-\varphi$.

Lemma 20.9.2 (Reliable sampling). *Let S be a set of n objects, $0 < \varepsilon < 1/2$, and let $\mathbf{r}$ be a point of depth $u \geq k$ in S. Let R be a random sample of S, such that every element is picked to be in the sample with probability*

$$p = \frac{8}{k\varepsilon^2}\ln\frac{1}{\delta}.$$

Let X be the depth of $\mathbf{r}$ in R. Then, we have that the estimated depth of $\mathbf{r}$ in R, that is, X/p, lies in the interval $[(1-\varepsilon)u, (1+\varepsilon)u]$. This estimate succeeds with probability $\geq 1-\delta^{u/k} \geq 1-\delta$.

Bibliography

[AC09] P. Afshani and T. M. Chan. *On approximate range counting and depth. Discrete Comput. Geom.*, 42(1): 3–21, 2009.

[AH08] B. Aronov and S. Har-Peled. *On approximating the depth and related problems*. *SIAM J. Comput.*, 38(3): 899–921, 2008.

[Ame94] N. Amenta. *Helly-type theorems and generalized linear programming*. *Discrete Comput. Geom.*, 12: 241–261, 1994.

[BG11] Y. Brise and B. Gärtner. *Clarkson's algorithm for violator spaces*. *Comput. Geom. Theory Appl.*, 44(2): 70–81, 2011.

[Cha05] T. M. Chan. *Low-dimensional linear programming with violations*. *SIAM J. Comput.*, 34(4): 879–893, 2005.

[Cha96] T. M. Chan. *Fixed-dimensional linear programming queries made easy*. *Proc. 12th Annu. Sympos. Comput. Geom.* (SoCG), 284–290, 1996.

[Cla88] K. L. Clarkson. *Applications of random sampling in computational geometry, II*. *Proc. 4th Annu. Sympos. Comput. Geom.* (SoCG), 1–11, 1988.

[Cla95] K. L. Clarkson. *Las Vegas algorithms for linear and integer programming*. *J. Assoc. Comput. Mach.*, 42: 488–499, 1995.

[CS89] K. L. Clarkson and P. W. Shor. *Applications of random sampling in computational geometry, II*. *Discrete Comput. Geom.*, 4: 387–421, 1989.

[ERK96] H. Everett, J.-M. Robert, and M. Kreveld. *An optimal algorithm for the* $(\leq k)$*-levels, with applications to separation and transversal problems*. *Int. J. Comput. Geom. Appl.*, 6(3): 247–261, 1996.

[GMRŠ06] B. Gärtner, J. Matoušek, L. Rüst, and P. Škovroň. *Violator spaces: Structure and algorithms*. *Proc. 14th Annu. Euro. Sympos. Alg.* (ESA), 387–398, 2006.

[GMRŠ08] B. Gärtner, J. Matoušek, L. Rüst, and P. Škovroň. *Violator spaces: Structure and algorithms*. *Discrete Appl. Math.*, 156(11): 2124–2141, 2008.

[Gol95] M. Goldwasser. *A survey of linear programming in randomized subexponential time*. *SIGACT News*, 26(2): 96–104, 1995.

[Grü03] B. Grünbaum. *Convex polytopes*. 2nd. Prepared by V. Kaibel, V. Klee, and G. Ziegler. Springer, May 2003.

[Har11] S. Har-Peled. *Geometric approximation algorithms*. Vol. 173. Math. Surveys & Monographs. Boston, MA, USA: Amer. Math. Soc., 2011.

[Har16] S. Har-Peled. *Shortest path in a polygon using sublinear space*. *J. Comput. Geom.*, 7(2): 19–45, 2016.

[HW87] D. Haussler and E. Welzl. *ε-nets and simplex range queries*. *Discrete Comput. Geom.*, 2: 127–151, 1987.

[Kal92] G. Kalai. *A subexponential randomized simplex algorithm*. *Proc. 24th Annu. ACM Sympos. Theory Comput.* (STOC), 475–482, 1992.

[Mat95] J. Matoušek. *On geometric optimization with few violated constraints*. *Discrete Comput. Geom.*, 14: 365–384, 1995.

[Meg83] N. Megiddo. *Linear-time algorithms for linear programming in* $\boldsymbol{R}^3$ *and related problems*. *SIAM J. Comput.*, 12(4): 759–776, 1983.

[Meg84] N. Megiddo. *Linear programming in linear time when the dimension is fixed*. *J. Assoc. Comput. Mach.*, 31: 114–127, 1984.

[MSW96] J. Matoušek, M. Sharir, and E. Welzl. *A subexponential bound for linear programming*. *Algorithmica*, 16(4/5): 498–516, 1996.

[Rüs07] L. Y. Rüst. *The P-Matrix Linear Complementarity Problem – Generalizations and Specializations*. Diss. ETH No. 17387. PhD thesis. ETH, 2007.

[Sei91] R. Seidel. *Small-dimensional linear programming and convex hulls made easy*. *Discrete Comput. Geom.*, 6: 423–434, 1991.

[Ško07] P. Škovroň. *Abstract models of optimization problems*. http://kam.mff.cuni.cz/~xofon/thesis/diplomka.pdf. PhD thesis. Charles University, 2007.

[SW92] M. Sharir and E. Welzl. *A combinatorial bound for linear programming and related problems*. *Proc. 9th Annul. Sympos. Theoret. Asp. Comput. Sci.* (STACS), vol. 577. 569–579, 1992.

[Van97] R. J. Vanderbei. *Linear programming: foundations and extensions*. Kluwer, 1997.