

Chapter 11

Linear programming in Low Dimensions

By Sarel Har-Peled, February 1, 2010^①

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

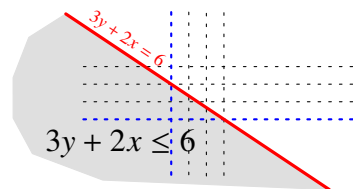
In this chapter, we shortly describe (and analyze) a simple randomized algorithm for linear programming in low dimensions. Next, we show how to extend this algorithm to solve linear programming with violations. Finally, we would show how one can efficiently approximate the number constraints one need to violate to make a linear program feasible. This serves as a fruitful ground to demonstrate some the techniques we visited already.

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

11.1 Linear Programming

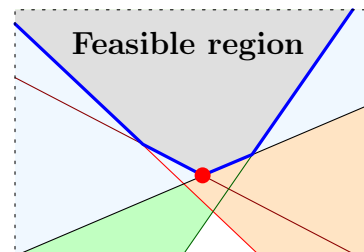
Assume we are given a set of n linear inequalities defined of the form $a_1x_1 + \dots + a_dx_d \leq b$, where a_1, \dots, a_d, b are constants, and x_1, \dots, x_d are the variables. In the **linear programming** (LP) problem, one has to find a **feasible solution**; that is, a point (x_1, \dots, x_d)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

The set of points complying with a linear inequality $a_1x_1 + \dots + a_dx_d \leq b$, is just a halfspace of \mathbb{R}^d , having the hyperplane $a_1x_1 + \dots + a_dx_d = b$ as a boundary, see figure on the right. As such, the feasible region of the LP is the intersection of n halfspaces; that is, it is a **polyhedron**. 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**.



^①This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

For the sake of simplicity of exposition, it would be easiest to think on the direction that one has to optimize for as the negative x_d -axis direction. This can be easily realized by rotating 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 vessel (with a convex interior). Next, we release a ball at the top of vessel, and the ball roll down (by “gravity” in the direction of the negative x_d -axis) till it reaches the lowest point in the vessel and get “stuck”. This point is the optimal solution to the LP that we are interested in computing.



In the following, we will assume that the given LP is in general position. Namely, if we intersect k hyperplanes, induced by k inequalities in the given LP, then their intersection is $d - k$ dimensional affine subspace. In particular, intersection of d of them is a point (referred to as a *vertex*). Similarly, intersection of any $d + 1$ of them is empty.

A polyhedron defined by a LP with n constraints might has $O(n^{\lfloor d/2 \rfloor})$ vertices on its boundary (this is known as the upper-bound theorem [?]). 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.

The fact we are interested in the best vertex of the feasible region, while this polyhedron is defined implicitly as the intersection of halfspaces also hints into 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 mentioned before, we will prove correctness in the next chapter).

11.1.1 A solution, and how to verify it

Observe that an optimal solution of a LP 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 provide 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 intersection the boundary of the polyhedron by a hyperplane), we can move it so that it lies on a $(k - 1)$ -dimensional facet F' of the feasible polyhedron, using the proceedings argumentation. Using it repeatedly, one ends up in a vertex of the polyhedron, or in an unbounded solution.

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

- (A) **Finite.** The optimal solution is finite, and the solver would provides a vertex which realizes the optimal solution.
- (B) **Unbounded.** The given LP has an unbounded solution. In this case, the LP would output a ray ζ , such that the ζ lies inside the feasible region, and it points downward the negative x_d -axis direction.

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

Lemma 11.1.1 *Given a set of d linear inequalities in \mathbb{R}^d , one can compute the vertex formed by the intersection of their boundaries in $O(d^3)$ time.*

Proof: Write down the system of equalities that the vertex must fulfil. Its a system of d equalities in d variables and it can be solved in $O(d^3)$ 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 $d - 1$ of the hyperplanes of a basis form a line and clipping this line to the cone of the basis form a ray. Clipping the same line to the feasible region would yield either a segment, referred to as an **edge** of the polytope, or a ray. 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 are the vertices of the polyhedron, and the edges of the polyhedron. Since every vertex has d hyperplanes defining it (its basis), and an adjacent edge is defined by $d - 1$ of these hyperplanes, it follows that each vertex has $\binom{d}{d-1} = d$ 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 how the feasible region of a linear program looks like, and we delegate it to the next chapter.

Lemma 11.1.2 *Let \mathcal{L} be a given linear program, and let \mathcal{P} denote its feasible region. Let \mathbf{v} be a vertex \mathcal{P} , such that all the d rays emanating from \mathbf{v} are in the upward x_d -axis direction, then \mathbf{v} is the lowest (in the x_d -axis direction) point in \mathcal{P} and it is thus the optimal solution to \mathcal{L} .*

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

Lemma ?? offers a simple algorithm for computing the optimal solution for an LP. Start from a feasible vertex of the LP. As long as this vertex has at least one ray that points downward, follow this ray to 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 ?? this is the optimal solution. Up to tedious (and non-trivial) details this is the **simplex** algorithm.

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

Lemma 11.1.3 *If \mathcal{L} is a LP in d dimensions which is not feasible, then there exists $d+1$ inequalities in \mathcal{L} which are infeasible on their own.*

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

11.2 Low Dimensional Linear Programming

11.2.1 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.

We remind the reader that the input to the algorithm is the LP \mathcal{L} which is defined by a set of n linear inequalities in \mathbb{R}^d . We are looking for the lowest point in \mathbb{R}^d which is feasible for \mathcal{L} .

A vertex v is **acceptable** if all the d rays associated with it points upward (note, that the vertex itself might not be feasible). The optimal solution (if it is finite) must be located at an acceptable vertex. Assume that we are given the basis $B = \{h_1, \dots, h_d\}$ of such an acceptable vertex. Let h_{d+1}, \dots, h_m be a random permutation of the remaining constraints of the LP \mathcal{L} .

Our algorithm is randomized incremental. At the i th step, for $i \geq d$, it would maintain the optimal solution for the first i constraints. As such, in the i th 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 $v_i \leftarrow v_{i-1}$.

The more interesting case, is when $v_{i-1} \notin h_i$. First, we check if the basis of v_{i-1} together with h_i form a set of constraints which is infeasible. If so, the given LP is infeasible, and we output $B(v_{i-1}) \cup \{h_i\}$ as our 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 $(d-1)$ -dimensional polyhedron $(\partial h_i) \cap \bigcap_{j=1}^{i-1} h_j$. This is a linear program involving $i-1$ constraints, and it involves $d-1$ variables since it lies on the $(d-1)$ -dimensional hyperplane ∂h_i . The solution found v_i is defined by a basis of $d-1$ constraints, and adding h_i to it, results in an acceptable vertex that is feasible, and we continue to the next iteration.

Clearly, the vertex v_n is the required optimal solution.

11.2.1.1 Running time analysis

Checking if a set of $d+1$ constraints is infeasible takes $O(d^4)$ time. The bad case for us, is when the vertex v_i is recomputed in the i th iteration. But 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 base, and there are at least $i-d$ constraints that are being randomly ordered (as the first d slots are fixed), we have that the probability that $v_i \neq v_{i-1}$ is

$$\alpha_i \leq \min\left(\frac{d}{i-d}, 1\right) \leq \frac{2d}{i},$$

for $i \geq d+1$, as can be easily verified.^② So, let $T(m, d)$ be the expected time to solve an LP with n constraints in d dimensions, we have

$$T(m, d) \leq O(md^3) + \sum_{i=d+1}^m \alpha_i(di + T(i, d-1)) = O(md^3) + \sum_{i=d+1}^m \frac{2d}{i} T(i, d-1).$$

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

Guessing that $T(n, d) \leq c_d n$, we have that

$$T(m, d) \leq c_1 m d^3 + \sum_{i=d+1}^m \frac{2d}{i} c_{d-1} i = (c_1 d^3 + 2d c_{d-1}) m,$$

where c_1 is some absolute constant. We need that

$$c_1 d^3 + 2c_{d-1} d \leq c_d,$$

Which holds for $c_d = O((3d)^d)$, and $T(m, d) = O((3d)^d m)$.

Lemma 11.2.1 *Given an LP with n constraints in d dimensions, and an acceptable vertex for this LP, then can compute the optimal solution in expected $O((3d)^d n)$ time.*

11.2.2 The algorithm for the general case

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

If the recursive call on $\widehat{\mathcal{L}} \cap h$ returned no solution, then the d constraints that prove that the LP $\widehat{\mathcal{L}}$ is infeasible on h , corresponds to a basis in \mathcal{L} of a vertex which is acceptable. Indeed, as we move these d constraints to the origin, their intersection is empty with h (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 ?? to solve the given LP.

If there is a solution to $\widehat{\mathcal{L}} \cap h$, then it is a vertex v on h which is feasible. Thus, consider the original set of $d-1$ constraints in \mathcal{L} that corresponds to the basis B of v . Let ℓ be the line formed by the intersection of the hyperplanes of B . Its 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 LP is unbounded.

Theorem 11.2.2 *Given a LP with n constraints defined over d variables, it can be solved in expected $O((3d)^d m)$ time.*

Proof: The expected running time is

$$S(m, d) = O(md) + S(m, d-1) + T(m, d),$$

where $T(m, d)$ is the time to solve a LP in the restricted case of Section ?. The solution to this recurrence is $O((3d)^d m)$, see Lemma ?. ■

11.3 Linear Programming with Violations

Let \mathcal{L} be a linear program with d variables, and $k > 0$ be a parameter. We are interested in the optimal solution of \mathcal{L} if we are allowed to throw away k constraints. A naive solution would be

to try and throw away all possible subsets of k constraints, solve each one of these instances and return the best solution found. This would require $O(n^{k+1})$ time. Luckily, it turns out that one can do much better if the dimension is small enough.

The idea is the following: The vertex realizing the optimal k -violated solution is a vertex v defined by d constraints (let its basis be B), and is of depth k . We remind the reader that a point p has **depth** k (in \mathcal{L}), if it is outside k halfspaces of \mathcal{L} (namely, we complement each constraint of \mathcal{L} , and p is contained inside k of these complemented hyperplanes). As such, we can use the depth estimation technique we encountered before. Specifically, if we pick each constraint of \mathcal{L} into a new instance of LP with probability $1/k$, then the probability the new instance $\widehat{\mathcal{L}}$ would have all the elements of B in its random sample, and will not contain any of the k constraints opposing v is

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

since $1 - x \geq e^{-2x}$, for $0 < x < 1/2$. If this happens then the optimal solution for $\widehat{\mathcal{L}}$ is v . This can be verified by computing how many constraints of \mathcal{L} the optimal solution of $\widehat{\mathcal{L}}$ violates. If it violates more than k constraints we ignore it. Otherwise, we return this as our candidate solution.

Next, we amplify the probability of success by repeating this process $M = 8k^d \ln(1/\delta)$ 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

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

Theorem 11.3.1 *Let \mathcal{L} be a linear program with n constraints over d variables, let $k > 0$ be a parameter, and $\delta > 0$ a confidence parameter. Then one can compute the optimal solution to \mathcal{L} violating at most k constraints of \mathcal{L} , in $O(mk^d \log(1/\delta))$ time. The solution returned is correct with probability $\geq 1 - \delta$.*

11.4 Approximate Linear Programming with Violations

The magic of Theorem ?? 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 a LP with n constraints (defined over d variables) the algorithm for computing 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 approximate solution in near linear time.

Lemma 11.4.1 *Let \mathcal{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 \mathcal{L} violating at most $(1 + \varepsilon)k$ constraints of \mathcal{L} such that its value is better than the optimal solution violating k constraints of \mathcal{L} . The expected running time of the algorithm is*

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

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

The algorithm succeeds with high probability.

Proof: Let $\rho = O\left(\frac{d}{k\varepsilon^2} \ln n\right)$ and pick each constraints of \mathcal{L} into $\widehat{\mathcal{L}}$ with probability ρ . Next, the algorithm computes optimal solution u in $\widehat{\mathcal{L}}$ violating

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

‘constraints, and return this as the required solution.

We need to prove the correctness of this algorithm. To this end, the reliable sampling lemma (Lemma ??) states that for any vertex v of depth u in \mathcal{L} , has depth in the range

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

in $\widehat{\mathcal{L}}$, and this holds with high probability, where $u \geq k$ (here we are using the fact that there are at most n^d vertices defined by \mathcal{L}).

In particular, let v_{opt} be the optimal solution for \mathcal{L} of depth k . With high probability, v_{opt} has depth $\leq (1 + \varepsilon/3)\rho k = k'$ in $\widehat{\mathcal{L}}$, which implies that the returned solution v is better than v_{opt} , since v has depth k' in $\widehat{\mathcal{L}}$.

Next, we need to prove that v is not too deep. So, assume that v is of depth β in \mathcal{L} . By the reliable sampling lemma, we have that the depth of v in $\widehat{\mathcal{L}}$ is in the range $\left[(1 - \varepsilon/3)\beta\rho, (1 + \varepsilon/3)\beta\rho\right]$. In particular, we know that $(1 - \varepsilon/3)\beta\rho \leq k' = (1 + \varepsilon/3)\rho k$. That is

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

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 ??. The input size is $O(n\rho)$ and the depth threshold is k' . (The bound on the input size holds with high probability. We omit the easy but the tedious proof of that using Chernoff inequality.) As such, the running time is

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

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

11.5 LP-type problems

Interestingly, the above algorithms 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 $G \subset \mathcal{H}$ 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}^d , and $w(G)$ is the radius of the smallest ball containing all the points of $F \subseteq \mathcal{H}$. 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:

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

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 $h \in H$, if

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

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

Definition 11.5.1 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. These operations are:

(A) (**Violation test.**) For a constraint h and a basis B , test whether h is violated by B or not. Namely, test if $w(B \cup \{h\}) > w(B)$.

(B) (**Basis computation.**) For a constraint h , and a basis B , computes the basis of $B \cup \{h\}$.

We also need to assume 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 s basis of \mathcal{H} . Its easy to verify that the algorithm we presented for linear programming (the special case of Section ??) works verbatim in this settings. Indeed, start with B_0 , and randomly permute the remaining constraints. Now, add the constraints in a random order, and each step check if the new constraints violates the current solution, and if so, update the basis of the new solution. The recursive call here, corresponds to solving a subproblem where some members of the basis are fixed. We conclude:

Theorem 11.5.2 *Let (\mathcal{H}, w) be a LP-type problem with n constraints with combinatorial dimension d . Assume that the basic operations takes constant time, we have that (\mathcal{H}, w) can be solved using $d^{O(d)}m$ basic operations.*

11.5.1 Examples for LP-type problems

Smallest enclosing ball. Given a set P of n points in \mathbb{R}^d , and let $r(P)$ denote the radius of the smallest enclosing ball in \mathbb{R}^d . Under general position assumptions, there are at most $d + 1$ points on the boundary of this smallest enclosing ball. We claim that the 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 $d + 1$. 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 to P . Thus, this is a LP-type problem, and it can be solved in linear time.

Theorem 11.5.3 *Given a set P of n points in \mathbb{R}^d , one can compute its smallest enclosing ball in (expected) linear time.*

Finding time of first intersection. Let $C(t)$ be a parameterized convex shape in \mathbb{R}^d , such that $C(0)$ is empty, and $C(t) \subseteq C(t')$ if $t < t'$. We are given n such shapes C_1, \dots, 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) $d + 1$ 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 also assume that these shapes are well behaved in the sense that, for any t , we have $\lim_{\Delta \rightarrow 0} \text{Vol}(C(t + \Delta) \setminus C(t)) = 0$ (namely, such a shape can not “jump” – it grows continuously). It is easy to verify that this is a 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 group a ball of radius t around 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.

11.6 Bibliographical notes

History. Linear programming has a rich and fascinating history. It can be traced back to the early 19th century. It started in earnest in 1939 when L. V. Kantorovich noticed the importance of certain type of Linear Programming problems. Unfortunately, for several years, Kantorovich work was unknown in the west and unnoticed in the east.

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 provide the right model for the analysis of classical economic theories. In 1975, both Koopmans and Kantorovich got the Nobel prize of economics. Dantzig probably did not get it because his work was too mathematical. So it goes.

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”). This makes however the usual description of the simplex algorithm pretty mysterious and counter-intuitive. 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 this algorithms, and the descriptions of the simplex algorithms usually detail how to handle these cases. See the book by Vanderbei [?] for an accessible and readable coverage 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 [?, ?]. His algorithm was deterministic but considerably more complicated than the randomized algorithm we present. Clarkson [?] showed how to use randomization to get a simple algorithm for linear programming with running time $O(d^2n + \text{noise})$, where the noise is a constant exponential in d . Our presentation follows the paper by Seidel [?]. Surprisingly, one can achieve running time with the noise being subexponential in d . This follows by plugging in the subexponential algorithms of Kalai [?] or Matoušek *et al.* [?] into Clarkson algorithm [?]. The resulting algorithm has expected running time $O(d^2n + \exp(c \sqrt{d \log d}))$, for some constant c . See the survey by Goldwasser [?] for more details.

More information on Clarkson’s algorithm. Clarkson’s algorithm contains some interesting new ideas that are worth mentioning shortly. (Matoušek *et al.* [?] algorithm 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 . Thus, repeating this d times, at each stage solving the problem on the collected constraints from previous iteration, together with the current random sample, results in a set of $O(d\sqrt{n})$ constraints that contains the optimal basis. Now solve recursively the linear program on this (greatly reduced) set of constraints. Namely, we spent $O(d^2n)$ time (d times checking if the n constraints violates a given solution), called recursively d times on “small” subproblems of size (roughly) $O(\sqrt{n})$, 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 algorithm works by combining these two algorithms together.

Linear programming with violations. The algorithm of Section ?? seems to be new, although it is implicit in the work of Matoušek [?], which present a slightly faster deterministic algorithm. The first paper on this problem (in two dimensions), is due to Everett *et al.* [?]. This was extended by Matoušek to higher dimensions [?]. His algorithm relies on the idea of computing all $O(k^d)$ 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 allows insertion and deletions of constraints, see the work by Chan [?]. Chan [?] 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 [?], 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 [?].

LP-type problems. The notion of LP-type algorithm is mentioned in the work of Sharir and Welzl [?]. They also showed up that deciding if a set of (axis parallel) rectangles can be pierced by 3-points is a LP-type problem (quite surprising as the problem has no convex programming flavor). Our example of computing first intersection of growing convex sets, is motivated by the work of Amenta [?] 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.

11.7 Exercises

11.8 From previous lectures

Lemma 11.8.1 (Reliable sampling.) *Let S be a set of n objects, $0 < \epsilon < 1/2$, 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 into the*

sample with probability

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

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