Discrete Comput Geom 6:423–434 (1991)

Discrete & Computational Geometry
© 1991 Springer-Verlag New York Inc.

Small-Dimensional Linear Programming and Convex Hulls Made Easy*

Raimund Seidel

Computer Science Division, University of California at Berkeley,
Berkeley, CA 94720, USA

Abstract. We present two randomized algorithms. One solves linear programs involving m constraints in d variables in expected time $O(m)$. The other constructs convex hulls of n points in $\mathbb{R}^d$, $d > 3$, in expected time $O(n^{\lfloor d/2 \rfloor})$. In both bounds d is considered to be a constant. In the linear programming algorithm the dependence of the time bound on d is of the form $d!$. The main virtue of our results lies in the utter simplicity of the algorithms as well as their analyses.

1. Introduction

One of the more exciting achievements in the theory of linear programming was accomplished in a series of papers by Megiddo and by Dyer, in the beginning of the last decade [M1], [M2], [D1], [D2], who showed that if d, the number of variables in a linear program, is considered a constant, then the linear program can be solved in time that is linear in m, the number of its constraints. These new algorithms were extremely complex, and, unfortunately, the running time of these algorithms depended on d in a superexponential way: for Megiddo's original algorithm [M2] the dependence was doubly exponential; subsequently, this was somewhat improved by Clarkson and by Dyer to a dependence of the form 3^{d^2} [C1], [D2].

More recently a number of randomized algorithms have been proposed [DF], [C2], where the most interesting one, due to Clarkson, has a remarkable expected running time of $O(d^2m) + (\log m)O(d)^{d/2+O(1)} + O(d^4\sqrt{m}\log m)$. That algorithm is

* Large portions of the research reported here were conducted while the author visited DIMACS at Princeton University. The author was supported by NSF Presidential Young Investigator Award CCR-9058440. Email address: seidel @ harmony.berkeley.edu.

relatively straightforward, however, the analysis of its expected running time is somewhat involved.

In the first part of this paper we present an exceedingly simple linear programming algorithm whose expected running time is $O(d!\, m)$. The analysis of its expected complexity is completely elementary and matches the algorithm in its simplicity.

The second part of this paper concerns the problem of constructing the convex hull of n points in $\mathbb{R}^d$. For dimension $d \leq 3$ this problem was essentially solved by the late seventies [G], [PH]. For $d > 3$ a number of deterministic algorithms have been proposed [CK], [K], [Sw], [S1], [S2], where the best time bounds achieved were $O(n^{\lceil d/2 \rceil})$, if measured in terms of input size n only, or $O(n^2 + F \log n)$, if measured in terms of input size n and output size F, the number of faces produced [S1], [S2] (d is considered a constant here).

In this paper we are only concerned with the case where $d > 3$ is a constant and where the running time is to be bounded by the input size only. Here the main open question has been whether it is possible to achieve a bound of $O(n^{\lfloor d/2 \rfloor})$, which would be worst-case optimal as the convex hull of n points in $\mathbb{R}^d$ can have this many faces. So far there has been no success with deterministic algorithms. However, recently Clarkson and Shor [CS] proposed a randomized algorithm with $O(n^{\lfloor d/2 \rfloor})$ expected running time.

In this paper we propose another randomized algorithm with the same performance. Our algorithm is similar to the one of Clarkson and Shor in that it is incremental. However, we avoid having to maintain so-called conflict graphs, which simplifies our algorithm and allows a very straightforward and elementary analysis of its expected running time.

The analyses of the expected running times of both algorithms in this paper rely heavily on the same idea, which can be expressed as "analyze the algorithm as if it were running backwards, from output to input." This view has proved very useful and is more thoroughly exploited in a forthcoming paper [S3].

2. Linear Programming

Geometrically, linear programming amounts to the following: we are given a set $\mathscr{H}$ of m half-spaces and a vector c in $\mathbb{R}^d$, and we want to find an "*optimum vertex*" v of the polyhedron $P_{\mathscr{H}}$ formed by the intersection of the half-spaces in $\mathscr{H}$, so that v maximizes the linear functional specified by c; in other words, v must be contained in the tangent hyperplane of $P_{\mathscr{H}}$ whose outward normal is c.

Consider the following strategy for finding such an optimum vertex v: Choose and remove a random H from the m half-spaces in $\mathscr{H}$ to obtain the set $\mathscr{H}'$. Recursively compute the optimum vertex v' of $P_{\mathscr{H}'}$ (with respect to the direction c). From v' compute v as follows: If v' is contained in the half-space H, then clearly $v = v'$, and nothing needs to be done. Otherwise v must be contained in the bounding hyperplane h of the half-space H. As a matter of fact, if $\bar{c}$ is the orthogonal projection of c into h and $\overline{\mathscr{H}'} = \{G \cap h \mid G \in \mathscr{H}'\}$, then v is the optimum vertex for the $(d-1)$-dimensional linear program specified by the half-spaces $\overline{\mathscr{H}}'$ (of h) and the direction $\bar{c}$. Thus v can be determined by recursively solving a $(d-1)$-dimensional problem with $m-1$ constraints. (A one-dimensional problem can be solved straightforwardly in time proportional to the number of its constraints.)

This is the gist of our algorithm. A number of important details still need to be addressed: How does the recursive procedure bottom out? What happens if an "optimum vertex" does not exist because of infeasibility or unboundedness of the problem?

Let us at first dispose of the unboundedness case. We stipulate that we are not interested in all of $\mathbb{R}^d$ but just some bounding box B_α (i.e., we impose explicit lower and upper bounds $-\alpha \le x_i \le \alpha$ on the d variables x_i). This bounding box also provides a convenient way for dealing with the bottoming out problem: if $\mathscr{H}$ is empty, then the optimum solution is one of the vertices of B_α and it can be determined from the signs of the coordinates of c in $O(d)$ time. Finally, infeasibility of the linear program (i.e., emptiness of $P_{\mathscr{H}}$) is discovered when the recursion has descended to the one-dimensional case.

What would the expected running time of our algorithm be? Why *expected* running time? Recall that the algorithm starts by choosing a half-space H from $\mathscr{H}$ at random. The expectation of the running time is to be taken with the assumption that whenever such a random choice is made each member of $\mathscr{H}$ is chosen equally likely.

For the sake of analysis let us assume at first that our linear programming problem and all the subproblems encountered are well behaved in the sense that the optimum vertex is unique, and that it is the intersection of the bounding hyperplanes of exactly d of the given half-spaces.

We claim that under these nondegeneracy assumptions our proposed method has an expected running time of $O(d!\, m)$. The proof, by induction on d, proceeds in a nutshell as follows: In case $d = 1$ the problem can be solved trivially in $O(m)$ time. For $d > 1$ it suffices to show that the expected time necessary to obtain v from v' is $O(d!)$. The interesting and expensive case happens when v' and v are different. But note that these vertices can only be different if one of the d half-spaces whose bounding hyperplanes define v is H. Since H was chosen from the m half-spaces in $\mathscr{H}$ uniformly at random, it follows that v is different from v' with probability d/m (at most, since some of the d hyperplanes defining v might derive from the bounding box B_α). By inductive assumption the expected cost of solving the ensuing $(d-1)$-dimensional linear program with $m-1$ constraints is $O((d-1)!\,(m-1))$. Thus the expected cost of obtaining v from v' is $(d/m)\cdot O((d-1)!\,(m-1))$, which is $O(d!)$, as claimed.

What about our nondegeneracy assumptions? We enforce uniquencess of the optimum vertex v by requiring that it be the vertex of $B_\alpha \cap P_{\mathscr{H}}$ that maximizes the inner product with c and that has the lexicographically largest coordinate representation. Note that for the analysis of the running time of the algorithm it is crucial that v is defined uniquely and canonically with respect to $\mathscr{H}$.

Finally, the assumption that v be the intersection of the bounding hyperplanes of exactly d half-spaces can be dropped altogether. However many half-spaces of $\mathscr{H}$ are involved in the definition of v, among them there can be at most d

half-spaces H with the property that the optimum vertex for $\mathscr{H}\backslash\{H\}$ is different from v.

We summarize:

Theorem 1. *Using the randomized method outlined in this section a linear program with m constraints in d variables can be solved in expected time $O(d!\,m)$.*

Proof. We just tidy up the analysis of the expected running time of our procedure. Assuming that testing whether a point is contained in a half-space takes $O(d)$ time, that projecting a d-vector orthogonally into a hyperplane takes $O(d)$ time, and that determining the intersection of a half-space in $\mathbb{R}^d$ with a hyperplane takes $O(d)$ time also, the expected running time $T(d, m)$ for our procedure satisfies

$$T(d,m) \le \begin{cases} O(m) & \text{if } d = 1, \\ O(d) & \text{if } m = 1, \\ T(d, m-1) + O(d) + \dfrac{d}{m} O(dm) + \dfrac{d}{m} T(d-1, m-1) & \text{otherwise.} \end{cases}$$

It is now easy to check that $T(d, m) = O(\sum_{1 \le i \le d}(i^2/i!)d!\,m)$, which is $O(d!\,m)$ since the sum converges even without an upper bound for i. □

The reader might object to our method of enforcing boundedness by imposing explicit lower and upper bounds on the variables. The number α might be chosen too small so that the bounding box B_α does not contain the optimum vertex of $P_{\mathscr{H}}$; or it might also be important to determine whether $P_{\mathscr{H}}$ is unbounded in the objective direction c.

There are at least two ways of dealing with such problems. One approach would be to amend the notion of "optimum solution" for a linear program: if $P_{\mathscr{H}}$ is bounded in the c-direction, then, as before, the optimum solution is a canonical vertex of $P_{\mathscr{H}}$ that maximizes the inner product with c; otherwise the optimum solution is a canonical direction in the recession cone of $P_{\mathscr{H}}$ for which the unit vector maximizes the inner product with c.

Another approach would be to continue using a bounding box B_α. However, we do not choose α explicitly but we use for α an indeterminate number λ larger than any number that ever appears in the computation, and we deal with λ symbolically. It turns out that this way the coordinates of the intermediate and final results in the computation are degree-1 polynomials in λ. In particular, the final optimum vertex is presented as $v(\lambda) = u + \lambda \cdot w$, where u and w are d-vectors. For all sufficiently large values for λ, the vector $v(\lambda)$ is then the optimum vertex of $B_\lambda \cap P_{\mathscr{H}}$. This means that if w is the zero vector, then the problem is bounded and u is the optimum vertex of $P_{\mathscr{H}}$; otherwise there is some real number λ_0 so that the ray $\{v(\lambda) | \lambda \ge \lambda_0\}$ is contained in $P_{\mathscr{H}}$. We detail this approach in the Appendix.

3. Convex Hulls

This section concerns the construction of the convex hull of a set S of n points in $\mathbb{R}^d$. We are only interested in the case $n > d > 3$, and we assume that S is in nondegenerate position, i.e., no $d + 1$ points of S lie in a common hyperplane. Such nondegeneracy can easily be simulated with impunity using standard perturbations techniques [E, p. 185]. Nondegeneracy ensures that the convex hull of any subset of S is a simplicial polytope.

First some basics: let P be a simplicial d-polytope, let V be the vertex set of P, and let $n = |V|$. It is known that P can have at most $O(n^{\lfloor d/2 \rfloor})$ faces [Mc]. We call the $(d - 1)$-faces of P *facets* and the $(d - 2)$-faces *ridges*. Every facet is uniquely identified by the d-tuple of its vertices. Similarly, every ridge can be identified by a $(d - 1)$-tuple of vertices in V. Since every ridge is contained in precisely two facets we can represent the facial structure of P by its *facet graph* $\mathscr{F}(P)$, which has the facets of P as its nodes and two facets adjacent iff they share a common ridge of P. Note that for simplicial d-polytopes the facet graph is regular of degree d. Throughout this section when we talk about "constructing the convex hull P of V" we really mean constructing the facet graph $\mathscr{F}(P)$. Moreover, we are not particularly careful with the distinction between facet F of P, the node corresponding to F in the facet graph $\mathscr{F}(P)$, and the d-tuple of vertices in V that span F. The same holds for ridges of P, edges of $\mathscr{F}(P)$, and $(d - 1)$-tuples of defining vertices.

Let p be some point in $\mathbb{R}^d$ in nondegenerate position with respect to V. We call a facet F of P *visible* from p iff the hyperplane spanned by F separates P and p. We call F *obscured* otherwise. We call a face G of P visible from p iff it is only contained in facets of P that are visible from p. Obscured faces are defined analogously. We call G a *horizon face* with respect to x iff it is contained in some visible and some obscured facet.

This terminology allows a convenient characterization of the facial structure of the polytope $P' = conv(P \cup \{x\})$ in terms of the faces of P: no visible face of P is a face of P'; all obscured and all horizon faces of P are faces of P'; for each horizon face G of P the pyramid $conv(G \cup \{x\})$ is a face of P'; this yields all faces of P'.

This characterization justifies the following method for obtaining P' from P and x. As stated before, we assume here that the polytopes are represented by their facet graphs. Thus, to be more precise, the procedure outlined below is intended to compute $\mathscr{F}(P')$ from $\mathscr{F}(P)$ and x:

(i) Locate some facet F of P that is visible from x, or determine that no such facet exist, in which case x is contained in P and hence $P' = P$.
(ii) Determine the set of facets and ridges of P that are visible from x and determine all horizon ridges of P with respect to x. Delete all visible facets and ridges.
(iii) For each horizon ridge G of P generate the new facet $conc(G \cup \{x\})$ of P' (i.e., a new node for the facet graph).
(iv) Generate the new ridges of P' (i.e., the edges between the new nodes of the facet graph).

Let us ignore for the moment how step (i) of this procedure can be done and let us examine the other steps in more detail.

Step (ii) can clearly be implemented via a depth-first search through $\mathscr{F}(P)$ that starts at F so that the time necessary is proportional to the number of visible faces found. Since all those faces are deleted, and since each face can be deleted only once, the cost of this step can be charged to the creation of each deleted face, and thus in the amortized sense step (ii) incurs no cost at all.

Step (iii) is straightforward and can be completed in time proportional to N_x, the number of new facets created.

The number of new ridges created in step (iv) is proportional to N_x. How can they be found? For every new facet generated in step (iii) the $d - 1$ new ridges contained in it can be determined "locally." Radix sorting the $(d - 1)$-tuples of vertices (or rather vertex indices) that identify these ridges then allows us to match them up and to form the new edges of the facet graph $\mathscr{F}(P')$ in time proportional to $n + N_x$.

It follows that if we ignore the cost of step (i), the total amortized cost of this procedure is $O(n + N_x)$, where n is the number of vertices of P and N_x is the number of facets of P' that contain x.

Let us still defer the details of how to deal with step (i) and let us consider the following algorithm for constructing the convex hull of a set S of $n > d$ points in $\mathbb{R}^d$ in nondegenerate position:

1. Put the points of S in a random order $p_1, \ldots, p_n$.
2. Form the facet graph $\mathscr{F}(P_{d+1})$, where $P_{d+1} = conv\{p_1, \ldots, p_{d+1}\}$. (Note that this graph is simply the complete graph on $d + 1$ vertices.)
3. For $d + 1 < i \leq n$, using the insertion procedure outlined above, form the facet graph $\mathscr{F}(P_i)$ from $\mathscr{F}(P_{i-1})$, where $P_i = conv\{p_1, \ldots, p_i\}$.

What is the expected running time of this algorithm? Obviously the crux of the question is what is the expected running time of step 3? In particular, what is the expected cost of computing $\mathscr{F}(P_i)$ from $\mathscr{F}(P_{i-1})$? We know that it is $O(i + N_i)$, where N_i is the number of facets of P_i that contain p_i. So what we need to determine is the expected value of N_i. Assuming that step 1 generates each permutation with equal probability, every one of the $j \leq i$ vertices of P_i was added last (i.e., was p_i) with equal probability. Since each facet of P_i contains exactly d vertices and since P_i has at most $O(j^{\lfloor d/2 \rfloor})$ facets it follows that the expectation of N_i is at most $(d/j)\, O(j^{\lfloor d/2 \rfloor})$, which is $O(i^{\lfloor d/2 \rfloor - 1})$. It follows therefore that the expected cost of the ith iteration of our algorithm is $O(i^{\lfloor d/2 \rfloor - 1})$, which implies that the total expected cost of the entire algorithm is $\sum_{d+1<i\leq n} O(i^{\lfloor d/2 \rfloor - 1})$, which is $O(n^{\lfloor d/2 \rfloor})$.

This analysis still neglects the cost of step (i) of the updating procedure. Recall that this step must find one visible facet of the "old" polytope P_{i-1} or determine that no such facet exists. Note that this is really a crucial step. It is exactly this problem, for instance, that forces Clarkson and Shor to resort to conflict graphs in their incremental convex hull algorithm. However, there is a straightforward solution to this problem, since it is nothing but a linear program in d dimensions involving one constraint for each vertex of P_{i-1}. Of course, for fixed d this can be solved in $O(i)$ time and for $d > 3$ this cost is subsumed by the cost of the remaining steps of the update procedure. Thus the expected running time of our incremental randomized convex hull algorithm remains $O(n^{\lfloor d/2 \rfloor})$.

We summarize:

Theorem 2. *Using the algorithm outlined in this section the convex hull of n points in $\mathbb{R}^d$ can be constructed in expected time $O(n^{\lfloor d/2 \rfloor})$, for any fixed constant $d > 3$.*

4. Remarks

The problem of locating a facet of the d-polytop $P = conv\ S$ that is visible from a point x can actually be formulated as a linear program in $d - 1$ dimensions: we want to find a hyperplane that contains x and is tangent to P. This in effect will locate a horizon ridge G of P and one of the two facets that contain G must be visible from x.

In our presentation we swept one problem under the rug: How does one correlate the output of the linear programming problem to the facet graph? The linear program will just produce the $(d - 1)$-set of vertices that span G. It needs a little bit of work to get from such a set to the actual edge in the facet graph. However, this can be done within the given time bound; for instance as follows: Let the points of S be numbered $p_1, \ldots, p_n$ according to the used random permutation. We maintain a sorted array of all ridges that have been created in the course of the algorithm and we correlate the array entries of the currently existing ridges with the corresponding edges of the current facet graph. Each ridge is represented as an ordered $(d - 1)$-tuple of the points that span it, ordered by decreasing point index. The array is sorted lexicographically in increasing order. Now, if the linear program outputs a $(d - 1)$-set T of vertices, we sort T into decreasing order and then use binary search to locate the resulting $(d - 1)$-tuple in our array in logarithmic time. From this entry we can determine the desired edge of the current facet graph in constant time. Updating the array is easy: When we add point p_i to the current hull the insertion algorithm already produces a lexicographically sorted list of all new ridges that contain p_i. Since i at that point is the currently largest index we only have to append that list to our master array.

The scheme just outlined might be undesirable as it uses space $O(n^{\lceil d/2 \rceil})$ in the worst case since no ridge is ever deleted from the array. Worst-case space $O(n^{\lfloor d/2 \rfloor})$ could be achieved, however, by using, instead of the array, a balanced-tree scheme that allows deletion of pointed to nodes in constant amortized or expected time. Examples of such tree schemes are red–black trees [T] or randomized search trees [AS].

A few words about the probability that the running times of the algorithms presented in this paper differ substantially from their expectations: We have been unable to prove any interesting results in this direction for the convex hull algorithm. For the linear programming algorithm the variance turns out to be very large, and we have not been able to prove that the probability of the algorithm exceeding its expected running time by a constant factor tends to 0 as m tends to infinity. However, it is possible to prove something slightly weaker.

Let Z_d^m be a discrete random variable measuring the number of operations made by our linear programming algorithm when applied to an input with m constraints in $d < m$ variables. The measurement will be pretty rough. We assume that each recursive invocation of the algorithm takes one unit of time for $m > 0$ and $d > 1$ (plus the time for recursive subcalls, of course), and that an invocation with $d = 1$ takes m units of time. This random variable reflects the actual running time of the algorithm reasonably well in that they are proportional to each other within a fixed factor. (A factor of $O(d)$ is easy to see, actually a factor of $O(1)$ holds.)

Let $p_d^m(x)$ be the generating function for Z_d^m. If we slow down the algorithm slightly so that one more constraint is used in the possible $(d-1)$-dimensional subcall, an upper bound for this generating function can be defined recursively as follows:

$$p_d^m(x) = \begin{cases} x^m & \text{if } d = 1, \\ 1 & \text{if } d > 1 \text{ and } m = 0, \\ p_d^{m-1}(x) \cdot x \cdot \left(\left(1 - \frac{d}{m}\right) + \frac{d}{m} p_{d-1}^m(x)\right) & \text{if } d > 1 \text{ and } m > 0. \end{cases}$$

This can be rewritten as

$$p_d^m(x) = \begin{cases} x^m & \text{if } d = 1 \\ x^m \prod_{1 \le j \le m} \left(\left(1 - \frac{d}{j}\right) + \frac{d}{j} p_{d-1}^j(x)\right) & \text{if } d > 1. \end{cases}$$

Note that this definition relies on the fact that there is no dependence between the vaious random choices made in the course of the algorithm.

Now let c_1 be any positive real number and depending on this number define recursively, for each $d > 1$,

$$c_d = (1 + c_1)e^{dc_{d-1}} - 1.$$

Lemma 1. *Let $m > 0$ be fixed and let $\alpha_m = (1 + c_1)^{1/m}$. Then, for $1 \le j \le m$,*

$$p_d^j(\alpha_m) \le (1 + c_d)^{j/m} \le 1 + \frac{j}{m} c_d.$$

Proof. The second inequality is true since for any $x > 0$ and any positive $\varepsilon \le 1$ the inequality $(1 + x)^\varepsilon \le 1 + \varepsilon x$ holds. We prove the first inequality by induction on d. For $d = 1$ it is true with equality by definition. For $d > 1$ we first use the inductive assumption and then the inequality $(1 + x) \le e^x$ to obtain

$$p_d^j(\alpha_m) = \alpha_m^j \prod_{1 \le i \le j} \left(\left(1 - \frac{d}{i}\right) + \frac{d}{i} p_{d-1}^i(\alpha_m)\right) \le \alpha_m^j \prod_{1 \le i \le j} \left(1 + \frac{d}{m} c_{d-1}\right)$$
$$\le \alpha_m^j e^{j(d/m)c_{d-1}} = ((1 + c_1)e^{dc_{d-1}})^{j/m} = (1 + c_d)^{j/m}. \qquad \square$$

Lemma 1 is of interest because of the following easily provable fact.

Fact 1. *Let $q(z)$ be the generating function of a nonnegative integer random variable X, and let $a > 1$ be any real number and k be some positive integer. Then*

$$\Pr(X \geq k) \geq \frac{q(a)}{a^k}.$$

Bounding the tail of the distribution of the random variable Z_d^m is now easy:

Lemma 2. *Let $c_1 > 0$, let c_d be defined as above, and let k be a positive integer, then*

$$\Pr(Z_d^m \geq k) \leq \frac{1 + c_d}{(1 + c_1)^{k/m}}.$$

Proof. Setting the a of Fact 1 to $\alpha_m = (1 + c_1)^{1/m}$ and using the bound of Lemma 1 we obtain

$$\Pr(Z_d^m \geq k) \leq \frac{p_d^m(\alpha_m)}{\alpha_m^k} \leq \frac{1 + c_d}{\alpha_m^k} = \frac{1 + c_d}{(1 + c_1)^{k/m}}. \qquad \square$$

Recalling that the expected value of Z_d^m is $d!\, m$ we obtain the following:

Corollary 2.1. *For any fixed constant $c > 1$ and any function $b(m)$ the probability that Z_d^m exceeds its expected value by a factor of $b(m)$ is $O(c^{-d!b(m)})$.*

Corollary 2.2. *For any fixed constant $c' > 0$ the probability that Z_d^m exceeds its expected value by a factor of* log *m is $O(m^{-c'd!})$.*

By optimizing the choice of the number c_1 it is possible to get more explicit bounds for small d. For instance, for $d = 2$ the linear programming algorithm exceeds its expected running time by a factor of 10 with probability at most 6.5×10^{-12}, and by a factor of 20 with probability at most 5.8×10^{-35}. In the three-dimensional case the expected running time is exceeded by a factor of 20 with probability at most 1.4×10^{-18}.

Mike Hohmeyer at UC Berkeley has implemented a version of the linear programming algorithm. Running a five-dimensional example with 2000 non-redundant constraints (hyperplanes tangent to a common paraboloid) 15 times on a DEC 3100 workstation required a minimum execution time of 4.7 seconds and a maximum of 18.6 seconds with an average execution time of 10.2 seconds. Various heuristics for speeding up the algorithm are being investigated.

Acknowledgments

I would like to thank Emo Welzl for inspiring discussions. I am grateful to an anonymous referee for setting the standards high. Tetsuo Asano kindly pointed out an error in a previous version of this paper. Finally I want to thank Ricky Pollack for making me write this paper.

Appendix

Here we give a more detailed description of a possible implementation of the randomized linear programming algorithm. The reader be warned that this is not necessarily a practical implementation: issues of numerical stability are ignored; most likely we should replace one of the recursions by iteration; employing the indeterminate λ also slows down the algorithm.

Below we detail a function *LP* that takes as inputs a positive integer d, a d-vector $c = (c_1, \ldots, c_d)$, and a set A of $(d + 2)$-vectors. *LP* either returns a pair of d-vectors u, w that have the property that for all sufficiently large reals λ the d-vector $u + \lambda w$ is the lexicographically largest vector x that maximizes $\sum_{1 \le i \le d} c_i x_i$ subject to the parametrized constraints

$$\sum_{1 \le i \le d} a_i x_i \le a_{d+1} + a_{d+2}\lambda \qquad \text{for each} \quad a = (a_1, \ldots, a_{d+2}) \in A$$

and

$$-\lambda \le x_i \le \lambda \qquad \text{for} \quad 1 \le i \le d,$$

or *LP* determines eventual infeasibility, i.e., for all sufficiently large λ the parametrized constraints do not admit a common solution. ("For all sufficiently large reals λ" is to mean "for all $\lambda \ge \lambda_0$," where λ_0 is some real number.)

Note that the variable bounds $-\lambda \le x_i \le \lambda$ imply that for any λ the system either has an optimum solution or is infeasible. Unboundedness is impossible.

Of course we are really interested in solving (possibly unbounded) linear programs of the form maximize $\sum_{1 \le i \le d} c_i x_i$ subject to the nonparametrized constraints

$$\sum_{1 \le i \le d} a_i x_i \le a_{d+1} \qquad \text{for each} \quad a = (a_1, \ldots, a_{d+1}) \in A,$$

where A is a set of $(d + 1)$-vectors. This problem can be solved by our function *LP* if we extend every $a \in A$ to a $(d + 2)$-vector by appending 0 as the $(d + 2)$nd coordinate. In other words, formally all constraints are now parametrized as

$$\sum_{1 \le i \le d} a_i x_i \le a_{d+1} + 0 \cdot \lambda.$$

If a set A of such extended vectors is supplied to our function *LP* along with the dimension d and the objective vector c, then either *LP*(d, c, A) determines eventual infeasibility, in which case the original problem is infeasible, or *LP*(d, c, A) returns a pair of d-vectors (u, w) with $w = \vec{0}$, in which case the original problem is bounded and u is the optimal solution, or *LP*(d, c, A) returns a pair of d-vectors (u, w) with

$w \neq \vec{0}$, in which case the original problem is unbounded and there is some number λ_0 so that the ray $\{u + \lambda w | \lambda \geq \lambda_0\}$ is contained in the feasible region.

The function *LP* specified below uses the test "$\leq_L$" which, given real numbers p, q, r, s, is to decide whether for all sufficiently large λ we have $(p + \lambda q) \leq (r + \lambda s)$. Of course this just amounts to a lexicographic comparison between the pairs (q, p) and (s, r). The operations "$<_L$," "$\max_L$," and "$\min_L$" are to be understood and performed analogously.

```
LP(d, c, A)
Base case $d = 1$:
  let $(h + h'\lambda) = \min_L(\{(a_2/a_1 + (a_3/a_1)\lambda) | a \in A, a_1 > 0\} \cup \{(0 + 1 \cdot \lambda)\})$
  let $(l + l'\lambda) = \max_L(\{(a_2/a_1 + (a_3/a_1)\lambda | a \in A, a_1 < 0\} \cup \{(0 - 1 \cdot \lambda)\})$
  let $(z + z'\lambda) = \min_L\{(a_2 + a_3\lambda) | a \in A, a_1 = 0\}$
  if $(z + z'\lambda) <_L (0 + 0 \cdot \lambda)$ or $(h + h'\lambda) <_L (l + l'\lambda)$ then stop and report
      infeasibility
if $c \geq 0$ then return the pair of 1-vectors $((h), (h'))$
        else return the pair of 1-vectors $((l), (l'))$
Case $d > 1$:
  if $A$ is empty then return the pair of $d$-vectors $(u, w)$, where, for
      $0 \leq i \leq d$, $u_i = 0$, and $w_i = 1$ if $c_i \geq 0$ and $w_i = -1$ otherwise
  Select some $a \in A$ uniformly at random and remove it from $A$ to ob-
      tain $A'$.
  let $(u, w)$ be the pair of $d$-vectors returned by $LP(d, c, A')$
  if $((\sum_{1 \leq i \leq d} a_i u_i) + (\sum_{1 \leq i \leq d} a_i w_i)\lambda) \leq_L (a_{d+1} + a_{d+2}\lambda)$ then return $(u, w)$
  else (* $(u + \lambda w)$ violates the constraint $a$. *)
    let $1 \leq k \leq d$ be maximal such that $a_k \neq 0$
    if no such $k$ exists then stop and report infeasibility
  (* Eliminate variable $x_k$ from constraints in $A'$ and from $c$. *)
    let $\bar{A}' = \{(b - (b_k/a_k)a)$ with the $k$th component removed $| b \in A'\}$
    let $\bar{c} = (c - (c_k/a_k)\hat{a})$ with the $k$th component removed, where
        $\hat{a} = (a_1, \ldots, a_d)$
  (* Incorporate the constraints $-\lambda \leq x_k \leq \lambda$ into $A'$. *)
    let $f$ be the $(d + 2)$-vector with $f_k = f_{d+2} = 1$ and all other compo-
        nents 0
    let $g$ be the $(d + 2)$-vector with $g_k = g_{d+2} = -1$ and all other
        components 0
    let $\bar{A} = \bar{A}' \cup \{(f - (1/a_k)a), (g - (1/a_k)a)$ with the $k$th component
        removed$\}$
  (* Solve the $(d - 1)$-dimensional problem and "lift" the solution. *)
    let $(\bar{u}, \bar{w})$ be the pair of $(d - 1)$-vectors returned by $LP(d - 1, \bar{c}, \bar{A})$
    let $u$ be the $d$-vector obtained from $\bar{u}$ by inserting 0 as the $k$th
        component
    let $w$ be the $d$-vector obtained from $\bar{w}$ by inserting 0 as the $k$th
        component
    let $u_k = (1/a_k)(a_{d+1} - \sum_{1 \leq i \leq d} a_i u_i)$
    let $w_k = (1/a_k)(a_{d+2} - \sum_{1 \leq i \leq d} a_i w_i)$
    return the pair of $d$-vectors $(u, w)$
```

References

[AS] C. R. Aragon and R. G. Seidel, Randomized Search Trees, *Proc. 30th IEEE Symp. on Foundations of Computer Science* (1989), pp. 540–545.

[CK] D. R. Chand and S. S. Kapur, An Algorithm for Convex Polytopes, *J. Assoc. Comput. Mach.* **17** (1970), 78–86.

[C1] K. L. Clarkson, Linear Programming in $O(n3^{d^2})$ Time, *Inform. Process. Lett.* **22** (1986), 21–24.

[C2] K. L. Clarkson, Las Vegas Algorithms for Linear and Integer Programming when the Dimension is Small, Manuscript *(Oct. 1989)*; a preliminary version appeared in *Proc. 29th IEEE Symp. on Foundations of Computer Science* (1988), pp. 452–456.

[CS] K. L. Clarkson and P. W. Shor, Applications of Random Sampling in Computational Geometry, II, *Discrete Comput. Geom.* **4** (1989), 387–422.

[D1] M. E. Dyer, Linear Algorithms for Two- and Three-Variable Linear Programs, *SIAM J. Comput.* **13** (1984), 31–45.

[D2] M. E. Dyer, On a Multidimensional Search Technique and Its Applications to the Euclidean One-Centre Problem, *SIAM J. Comput.* **15** (1986), 725–738.

[DF] M. E. Dyer and A. M. Frieze, A Randomized Algorithm for Fixed-Dimensional Linear Programming, *Math. Programming* **44** (1989), 203–212.

[E] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York (1987).

[G] R. L. Graham, An Efficient Algorithm for Constructing the Convex Hull of a Finite Planar Set, *Inform. Process. Lett.* **1** (1972), 132–133.

[K] M. Kallay, Convex Hull Algorithms in Higher Dimensions, Manuscript (1981).

[Mc] P. McMullen, The Maximum Number of Faces of a Convex Polytope, *Mathematika* **17** (1971), 179–184.

[M1] N. Megiddo, Linear-Time Algorithms for Linear Programming in $\mathbb{R}^3$ and Related Problems, *SIAM J. Comput.* **12** (1983), 759–776.

[M2] N. Megiddo, Linear Programming in Linear Time when the Dimension is Fixed, *J. Assoc. Comput. Mach.* **31** (1984), 114–127.

[PH] F. P. Preparata and S. J. Hong, Convex Hulls of Finite Point Sets in Two and Three Dimensions, *Comm. ACM* **20** (1977), 87–93.

[S1] R. Seidel, A Convex Hull Algorithm Optimal for Point Sets in Even Dimensions, Technical Report 81-14, Department of Computer Science, University of British Columbia (1981).

[S2] R. Seidel, Constructing Higher-Dimensional Convex Hulls at Logarithmic Cost per Face, *Proc. 18th ACM Symp. on Theory of Computing* (1986), pp. 404–413.

[S3] R. Seidel, Backwards Analysis of Randomized Geometric Algorithms (Manuscript).

[Sw] G. Swart, Finding the Convex Hull Facet by Facet, *J. Algorithms* **6** (1985), 17–48.

[T] R. E. Tarjan, *Data Structures and Network Algorithm*, Society for Industrial and Applied Mathematics Philadelphia, PA (1983).

Received September 1, 1990, and in revised form March 4, 1991.