Linear optimization queries*

Jiří Matoušek†
Department of Applied Mathematics, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czechoslovakia

Otfried Schwarzkopf ‡
Utrecht University, Department of Computer Science
P.O. Box 80-089, 3508 TB Utrecht, the Netherlands

Abstract

Let Γ be a set of n halfspaces in E^d (where the dimension $d \geq 3$ is fixed) and c a d-component vector. We denote by $\mathrm{LP}(\Gamma, c)$ the linear programming problem of minimizing the function $c \cdot x$ over the intersection of all halfspaces of Γ. We show that Γ can be preprocessed in time and space $O(m^{1+\delta})$ (for any fixed $\delta > 0$, m is an adjustable parameter, $n \leq m \leq n^{\lfloor d/2 \rfloor}$) so that given $c \in \mathrm{E}^d$, $\mathrm{LP}(\Gamma, c)$ can be solved in time $O((\frac{n}{m^{1/\lfloor d/2 \rfloor}} + |\Gamma_q|)\log^{2d+1} n)$. The data structure can be dynamically maintained under insertions and deletions of hyperplanes from Γ, in $O(m^{1+\delta}/n)$ amortized time per update operation. We use a multidimensional version of Megiddo's parametric search technique.

In connection with an output-sensitive algorithm of Seidel, we get that a convex hull of an n-point set in E^d ($d \geq 4$) can be computed in time $O(n^{2-\frac{2}{1+\lfloor d/2 \rfloor}+\delta} + h\log n)$, where h is the number of faces of the convex hull. We also show that given an n-point set P in E^d, one can determine the extreme points of P in time $O(n^{2-\frac{2}{1+\lfloor d/2 \rfloor}+\delta})$ (for any fixed $\delta > 0$).

*This extended abstract combines a paper [Mat91d] of the first author with an improvement and simplification achieved by the second author in [Sch91].

†The research by J. M. was performed while he was visiting at School of Mathematics, Georgia Institute of Technology, Atlanta.

‡O. S. acknowledges support by the ESPRIT II Basic Research Action of the European Community under contract No. 3075 (project ALCOM). This research was done while he was employed at Freie Universität Berlin. Furthermore, part of this research was done while he visited INRIA-Sophia Antipolis.

8th Annual Computational Geometry, 6/92, Berlin, Germany
©1992 ACM 89791-518-6/92/0006/0016 ..... $1.50

1 Introduction and statement of results

This research was originally motivated by the following problem: Let P be an n-point set in E^d, where the dimension d is fixed; determine which points of P are extreme (we call a point $x \in P$ *extreme* if $\mathrm{conv}(P \setminus \{x\}) \neq \mathrm{conv}(P)$). In dimensions 2 and 3, this problem can be solved relatively easily in optimal time $O(n \log n)$ by computing the convex hull of P (see [PH77]). But if the dimension is at least 4, then the complexity of an explicit description of the convex hull of P may be too large, since $\mathrm{conv}(P)$ can have up to $\Omega(n^{\lfloor d/2 \rfloor})$ facets.

It is not difficult to see that the question "is x extreme in P?" can be formulated as a linear programming problem with d variables. Megiddo [Meg84] has shown that a linear program with d variables and n constraints can be solved in $O(n)$ time. In Megiddo's original solution, the constant of proportionality grows doubly exponentially with d. Subsequent solutions improved this to a single exponential growth ([Dye86], [Cla86]). Also randomized algorithms with linear expected running time are known; the best asymptotic dependence on the dimension has been achieved by Clarkson [Cla88a], and a particularly simple algorithm was given by Seidel [Sei90].

With linear programming in linear time, the extreme points can be found in $O(n^2)$ time. Recently Seidel and Welzl [SW90] improved this complexity to $O(n^{3/2}\log n)$ in dimension 4 and to $O(n^{2-c_d}\log^{O(1)} n)$ in any fixed dimension $d \geq 5$, where $1/c_d = O(\lfloor d/2 \rfloor !^2)$. They use a randomized divide-and conquer to partition the problem into suitable smaller subproblems, which are then solved by linear programming.

Here we improve their result. We describe a way to preprocess a point set P in E^d for fast answering of *separation queries*. Given a query point $x \in \mathrm{E}^d$, our algorithm either finds a hyperplane h containing x and such that all points of P lie in one of the open halfspaces bounded by h, or determines that no such hyperplane

exists.

Theorem 1.1 *Given an n-point set P in E^d and a parameter m, $n \leq m \leq n^{\lfloor d/2 \rfloor}$, one can preprocess P with space and preprocessing time $O(m^{1+\delta})$ for any fixed $\delta > 0$, so that separation queries can be answered in time $O(\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^{2d+1} n)$. The data structure can be maintained in $O(m^{1+\delta}/n)$ amortized time per insert/delete operation.*

With a small modification of this data structure and when the query time for n queries and the preprocessing time are balanced appropriately, we obtain

Theorem 1.2 *The extreme points of an n-point set in E^d can be found in time $O(n^{2-\frac{2}{1+\lfloor d/2 \rfloor}+\delta})$ (for any fixed $\delta > 0$).*

Our approach can be easily extended to handle *linear optimization queries*. Let Γ be a set of halfspaces in E^d and c a d-component vector. We denote by $\mathrm{LP}(\Gamma, c)$ the linear programming problem of minimizing the function $c \cdot x$ over the intersection of all halfspaces of Γ. We show that one can preprocess a given set Γ_0 of halfspaces so that for a given c, $\mathrm{LP}(\Gamma_0, c)$ can be solved quickly. We may even add a set Γ_q of additional constraints (halfspaces) to the query and solve $\mathrm{LP}(\Gamma_0 \cup \Gamma_q, c)$ quickly. Results of this type for dimensions 2 and 3 were given by Guibas, Stolfi and Clarkson [GSC87]; they show that if the query consists of c only, the solution can be found in $O(\log n)$ time, where $n = |\Gamma_0|$, with $O(n \log n)$ time preprocessing and $O(n)$ storage.

For a dynamic version of the problem (when updates on Γ_0 are allowed) and for dimension 3, Eppstein [Epp91] gave a data structure for the special case when the sequence of insertions/deletions is known in advance. For $d \leq 4$ and assuming a random sequence of updates, Mulmuley [Mul91] gave an algorithm which has a polylogarithmic query time with high probability, and with expected amortized update costs $O(\log n)$ for $d = 3$, resp. $O(n \log^2 n)$ for $d = 4$.

We have the following quantitative results:

Theorem 1.3 *One can preprocess a set Γ_0 of n halfspaces in E^d in $O(m^{1+\delta})$ (deterministic) time and space, for a parameter m, $n \leq m \leq n^{\lfloor d/2 \rfloor}$, and some fixed $\delta > 0$, so that given a query d-component vector c and a set Γ_q of additional halfspaces, the problem $\mathrm{LP}(\Gamma_0 \cup \Gamma_q, c)$ can be solved in time $O((\frac{n}{m^{1/\lfloor d/2 \rfloor}} + |\Gamma_q|) \log^{2d+1} n)$. The data structure can be dynamically maintained under insertions and deletions of hyperplanes from Γ, in $O(m^{1+\delta}/n)$ amortized time per update operation.*

This result can be applied to improve an output-sensitive convex hull algorithm. Seidel [Sei86] gave an output-sensitive algorithm which computes the convex hull of an n point set in E^d in time $O(n^2 + h \log n)$, where h is the number of faces of the convex hull. By a standard lifting transformation, this result also applies to computing Voronoi diagrams one dimension lower. By a direct application of the preceding theorem in Seidel's algorithm, we get

Theorem 1.4 *The convex hull of an n point set in E^d can be computed in time $O(n^{2-\frac{2}{1+\lfloor d/2 \rfloor}+\delta} + h \log n)$. The nearest-point Voronoi diagram and farthest-point Voronoi diagram of an n point set in E^{d-1} can be computed in $O(n^{2-\frac{2}{1+\lfloor d/2 \rfloor}+\delta} + h \log n)$ time, where h is the combinatorial complexity of the diagram.*

The above results are proved using two main ingredients. One of them is a multidimensional version of Megiddo's parametric search technique (a similar technique has been applied previously e.g. in [CSY87], [CM89], [NPT90]). The second one are data structures for so-called halfspace emptiness queries. We mainly use known data structures ([Mat91c], [Cla88b], [AM91]), but we add a new capability to these structures, namely to provide a witness point if the query halfspace is nonempty. This can be done using parametric search (see [AM92]), but we provide a simpler and more efficient solution. As a by-product, these augmented data structures can also be used for answering the following *ray shooting queries*: Given a set H of n hyperplanes in E^d, and a ray ρ with origin p lying *above all hyperplanes of H*, find the first hyperplane intersected by ρ (the additional requirement on p makes sure we only have to search within the upper envelope of H).

Theorem 1.5 *Given a set H of n hyperplanes in E^d ($d \geq 4$), and a parameter m ($n \leq m \leq n^{\lfloor d/2 \rfloor}$), there is a data structure for ray shooting queries in the upper envelope of H with space and preprocessing time $O(m^{1+\delta})$ and query time $O(n/m^{1/\lfloor d/2 \rfloor} \log n)$, and which supports dynamic insertions and deletions of hyperplanes with amortized update time $O(m^{1+\delta}/n)$.*

This improves a structure for ray shooting queries by [AM92]. There is a standard application of such a ray shooting result to nearest neighbor queries. Using the usual lifting map (see e.g. [Ede87]), nearest-neighbor queries in E^{d-1} are transformed into shooting a vertical ray in the upper envelope of a set of hyperplanes in E^d, and we get the corresponding slightly improved bounds for the nearest neighbor problem.

The time bounds given in Theorems 1.1, 1.3, and 1.5 above are for the most general situation. For the case of linear or maximal space, and if a dynamic structure is not necessary, we can get slightly better bounds,

summarized in Table 1. This table shows space and amortized update time for several data structures (the preprocessing time is only mentioned if it differs from space requirements), and the query times for query algorithms using these structures for ray shooting queries (or halfspace emptiness queries), and separation queries (or linear optimization queries) resp. In all cases, $\delta > 0$ is an arbitrarily small constant, and m is a parameter with $n \leq m \leq n^{\lfloor d/2 \rfloor}$. The three last structures do not support dynamic updates.

2 Halfspace emptiness and ray shooting queries

In this section, we will consider the following problem: Given an n-point set P in E^d, preprocess it in such a way that given a query halfspace γ, we can quickly determine whether γ contains some point of P. We will call this the *halfspace emptiness* problem (there are actually two versions of this problem, since we may consider an open halfspace or a closed one). We will describe some known results for this problem. However, the subsequent application of parametric search will require that the considered data structure for halfspace emptiness queries always provides some point lying in the query halfspace, provided that the halfspace is nonempty (we call such a point a *witness*).

There is a general technique (by [AM92]) for obtaining a witness using any algorithm for halfspace emptiness (satisfying certain mild requirements). This technique would bring some technical complications in our forthcoming parametric search applications, and it also loses some logarithmic factors in the asymptotic efficiency. Here we describe alternative and simpler ways for finding a witness. It turns out that this technique developed for finding a witness can handle also the more general ray shooting queries.

2.1 Algorithms with almost linear space

The best known results for the halfspace emptiness problem in dimension $d \geq 4$ with linear space were recently attained in [Mat91c]. A dynamization of this data structure is discussed in [AM91]. Neither of these data structures provides a witness directly.

We start with several definitions. Let P be an n-point set in E^d. A *simplicial partition* of P is a collection

$$\Pi = \{(P_1, \Delta_1), \ldots, (P_m, \Delta_m)\},$$

where the P_i's are nonempty sets (called the *classes* of Π) forming a partition of P and each Δ_i is a relatively open simplex (not necessarily full-dimensional) containing P_i. The number m is the *size* of the partition.

The basis of the efficient algorithm for the halfspace emptiness problem in [Mat91c] is the so-called Partition theorem for shallow hyperplanes. We say that a hyperplane h is *k-shallow* (relative to P), if one of the halfspaces bounded by h contains no more than k points of P in its interior. We say that a hyperplane h *crosses* a simplex Δ if $\Delta \cap h \neq \emptyset$ but $\Delta \not\subseteq h$. For our purposes here, we use the following form of the theorem:

Theorem 2.1 [Mat91c] (Partition theorem for shallow hyperplanes) *Let P be a set of n points in E^d, $d \geq 4$, and r a parameter, $1 \leq r < n$. Then there exists a simplicial partition Π of size $O(r)$ for P, whose classes have size between $\lfloor n/r \rfloor$ and $2n/r$, and such that any n/r-shallow hyperplane h crosses at most $O(r^{1-1/\lfloor d/2 \rfloor})$ simplices of Π. For $r \leq n^{\alpha}$, where $\alpha > 0$ is a certain constant (depending on the dimension d), such a partition can be found in time $O(n \log r)$.*

In this formulation we added one property compared to the version in [Mat91c] (proceedings version), namely the lower bound on the cardinality of the classes. This, however, follows directly by inspecting the proof in [Mat91c], and will appear in a journal version of that paper.

The preprocessing for a basic version of the halfspace emptiness algorithm consists of building a *partition tree* using the above theorem. Each node v of the tree corresponds to some subset $P_v \subseteq P$, the root corresponds to the whole set P. We choose a suitable large constant r, we find a simplicial partition of P of size $O(r)$ as in the above theorem and we store the description of the simplices of this partition in the root of the tree. For every subset P_i of this partition, we then build one subtree of the root in the same manner. The construction ends in the leaves of the tree, where the size of the subsets drops below some constant; then we explicitly store all the points of such a subset in the corresponding leaf.

The query answering algorithm deciding the emptiness of an open halfspace γ with a bounding hyperplane h then uses the partition tree as follows: We start in the root of the tree, and we determine the simplices of the simplicial partition stored there intersecting γ. If the number of such simplices is greater than the guaranteed maximal number $\kappa = O(r^{1-1/\lfloor d/2 \rfloor})$ of simplices crossed by any n/r-shallow hyperplane, then γ is nonempty, in fact in contains at least $\lfloor n/r \rfloor$ points of P (since either there is a simplex completely contained in γ, or more than κ simplices cross h, implying that h is not n/r-shallow). Otherwise we proceed recursively down the tree into the at most κ children of the current node for which the corresponding simplices intersect γ. This recursion terminates in leaves of the partition tree, where we simply check if any of the points stored there is contained in γ.

space; preprocessing	update time (amortized)	ray shooting queries	separation queries
n; $n \log n$	$\log^2 n$	$n^{1-1/\lfloor d/2 \rfloor + \delta}$	$n^{1-1/\lfloor d/2 \rfloor + \delta}$
$n^{\lfloor d/2 \rfloor + \delta}$	$n^{\lfloor d/2 \rfloor - 1 + \delta}$	$\log n$	$(\log n)^{d+1}$
$m^{1+\delta}$	$m^{1+\delta}/n$	$\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log n$	$\frac{n}{m^{1/\lfloor d/2 \rfloor}} (\log n)^{2d+1}$
n; $n \log n$	N/A	$n^{1-1/\lfloor d/2 \rfloor} (\log n)^{O(1)}$	$n^{1-1/\lfloor d/2 \rfloor} (\log n)^{O(1)}$
$n^{\lfloor d/2 \rfloor} (\log n)^{O(1)}$	N/A	$\log n$	$(\log n)^{d+1}$
$m (\log n)^{O(1)}$	N/A	$\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log n$	$\frac{n}{m^{1/\lfloor d/2 \rfloor}} (\log n)^{2d+1}$

Table 1: Summary of data structures

A straightforward analysis of this data structure shows that it requires $O(n)$ space, $O(n \log n)$ preprocessing time and $O(n^{1-1/d+\delta})$ query time, where $\delta > 0$ tends to 0 with increasing r. A dynamic version of this data structure is described in [AM91]. First, it is shown that one can perform deletions of points from this data structure (rebuilding subtrees from which many points have been deleted). Insertions are then handled in a standard way ("binary counting" pattern) using the decomposability of the halfspace emptiness problem. The resulting (amortized) update time is $O(\log^2 n)$.

We will now discuss several ways to find a witness using this data structure. The problematic situation is the following:

> In the query answering algorithm, we are at an inner node v, and we find that γ intersects more than κ simplices of the simplicial partition in v.

We say that such a halfspace γ is *deep* for v. By the argument above, we have $|\gamma \cap P_v| > n_v/r$, $n_v = |P_v|$, for any deep halfspace γ. In the dynamic version described in [AM91], however, we have to be a bit careful. Fortunately, we reconstruct a subtree when $n_v/2r$ points have been deleted from P_v, so we can conclude that for any deep halfspace γ, we have at least $|\gamma \cap P_v| > n_v/2r$.

A simple randomized algorithm. When we reach a node v for which γ is deep, we pick a random point p in P_v. By the above, the probability that $p \in \gamma$ is at least $1/2r$, which implies that the probability that we do not hit a point in γ after $O(r \log n)$ trials is very small (less than $1/n^c$, for a constant $c > 0$ that can be made arbitrarily large). A random point in P_v can be found in time $O(\log n_v)$ without having to store the set P_v explicitly: if we store the number of points in P_v with each node, we can just choose a random index in $\{1, \ldots, n_v\}$ and track down the point in the corresponding subtree, in $O(\log n_v)$ time. The time for witness finding is thus dominated by the total query time with high probability. We needed no auxiliary data structures (except for the point counts in nodes, whose maintenance is trivial), and so this query answering algorithm can be used for the dynamic version as well.

A deterministic algorithm (static version). The key to a deterministic query algorithm providing a witness is the notion of ε-nets. A *$(1/r)$-net* for an n-point set P is a subset $R \subset P$ with the property that every halfspace γ with $|\gamma \cap P| \geq n/r$ contains a point of R. We use the following lemma.

Lemma 2.2 (ε-net theorem for halfspaces) *Let P be a set of n points in E^d, $r \leq n$ a parameter. There exists a $(1/r)$-net R of P with $|R| = O(r \log r)$* [HW87], *and it can be computed in time $O(n \log r)$ time if $r \leq n^\alpha$, where $\alpha > 0$ is certain constant* [Mat91b].

We augment the (static) data structure for halfspace emptiness queries as follows: we add a $(1/r)$-net R_v of size $O(r \log r)$ for the set P_v to every inner node v. Queries proceed as before, but when we reach a node v for which γ is deep, we test all points $p \in R_v$ for containment in γ. Since γ is not n/r-shallow, $\gamma \cap R_v \neq \emptyset$, so we find a witness in additional time $O(|R_v|) = O(r \log r)$. Query time and storage of the structure thus remain the same as before, and this also holds for the preprocessing time (since R_v can be computed in time linear in the size of P_v).

For a static data structure, the above bounds can be slightly improved, as in [Mat91c]. Instead of setting r to a large enough constant, we can choose $r = n^\alpha$ for some sufficiently small constant $\alpha > 0$, and use exactly the same structure otherwise. The space is clearly still linear, the preprocessing time is $O(n \log n)$ and the query time $O(n^{1-1/\lfloor d/2 \rfloor} (\log n)^{O(1)})$, also when we count the additional witness finding time. We do not know how to dynamize this improved data structure efficiently, so in the next algorithm we will return to constant values of r at every node.

We also have a deterministic dynamic version of the algorithm. Details can be found in the report version [MS91].

Ray shooting algorithms. For a point set P and a nonvertical hyperplane h, let us call a point $p \in P$ *h-extreme* if there is no point of P above the hyperplane passing through p and parallel to h (let us denote this hyperplane by $h(p)$). Let us consider the problem of finding an h-extreme point of P for a query hyperplane h. This problem is intuitively more appealing in a dual setting, where we consider a collection of hyperplanes and we shoot vertical rays originating above all the hyperplanes.

It turns out that some of the data structures developed for halfspace emptiness queries with witness can be used to find an h-extreme point quite easily. Let us consider the data structure for the above presented static deterministic algorithm. Given a query hyperplane h, we start in the root of the data structure. In an inner node v, we proceed as follows: we find an h-extreme point $q \in R_v$ (R_v is the $(1/r)$-net at v), and we recursively find the h-extreme points for all children of v corresponding to simplices of Π_v intersecting the halfspace above $h(q)$ (if there is no such simplex, then q itself is h-extreme). Among these, we select an h-extreme one. In a leaf, an h-extreme point is selected trivially. This algorithm is obviously correct, and the $(1/r)$-net property of R_v guarantees that $h(q)$ is a n/r-shallow hyperplane (relative to P_v), and hence we recurse in at most κ children of v. Thus the running time analysis is the same as for the basic halfspace emptiness algorithm and we also get the same query time.

As we pointed out, the h-extreme point problem is equivalent to vertical ray shooting in an upper envelope of hyperplanes. One can generalize the solution to ray shooting with arbitrary rays originating above all the hyperplanes. In the primal setting, this corresponds to the following problem: we are given a query (nonvertical) $(d-2)$-flat a, such that there is a hyperplane h_0 passing through a and having no points of P above it. We want to find an *a-extreme point* of P, which is a point $p \in P$ such that there are no points of P above the hyperplane passing through p and a. The previously considered problem corresponds to the situation when a is formally at infinity. One can find an a-extreme point in almost exactly the same way as we did for an h-extreme point. Hence we can solve the ray shooting problem with the same complexity as the halfspace emptiness problem.

2.2 Algorithms with logarithmic query time

Another data structure for the halfspace emptiness problem is due to Clarkson [Cla88b]; it requires $O(n^{\lfloor d/2 \rfloor + \delta})$ space and preprocessing time, and achieves $O(\log n)$ query time. A dynamic counterpart of this data structure with the same asymptotic performance and $O(n^{\lfloor d/2 \rfloor - 1 + \delta})$ amortized update time was given in [AM91]. We will explain a static data structure (similar to Clarkson's one) and the way one finds a witness point using this structure.

We will work in a dual setting. Let H be the set of n hyperplanes dual to the points in P. Answering an empty half-space query for P reduces to determining whether a query point q lies above all hyperplanes of H; we will refer to the dual problem as the *upper envelope problem* for H.

Let H be a collection of hyperplanes in E^d, and let $r \leq n$ be a parameter. For the sake of simplicity, we will assume that the hyperplanes of H are in general position. In this situation, a collection Ξ of simplices with disjoint interiors will be called a *$(1/r)$-cutting for the (≤ 0)-level of H*, provided that the simplices of Ξ cover all points of level 0 (with respect to H, i.e. all points with no hyperplanes of H lying strictly above them), and that each simplex of Ξ is intersected by at most n/r hyperplanes of H. We will apply the following result:

Theorem 2.3 (Shallow cutting lemma, [Mat91c]) *Let H, r be as above. Then there exists a $(1/r)$-cutting Ξ for the (≤ 0)-level of H, consisting of $O(r^{\lfloor d/2 \rfloor})$ simplices. For $r \leq n^{\alpha}$ (where $\alpha > 0$ is a certain constant, dependent on the dimension), such a cutting can be computed in $O(n \log r)$ time.*

For every simplex $\Delta \in \Xi$, we say that a hyperplane $h \in H$ is *relevant* for Δ if it lies above Δ or intersects Δ. Let H_Δ denote the collection of hyperplanes relevant for Δ.

A slightly modified Clarkson's structure for the upper envelope problem is a tree-like structure, built recursively as follows: If the number of hyperplanes in H is smaller than a suitable constant, then one simply stores the list of hyperplanes of H; this will be a leaf node. A query is answered by testing the query point against each hyperplane of H.

If, on the other hand, H is large, one chooses a suitable parameter r (a sufficiently large constant in Clarkson's original construction), and computes a $(1/r)$-cutting Ξ for the (≤ 0)-level of H, consisting of $O(r^{\lfloor d/2 \rfloor})$ simplices. We store the cutting Ξ in the root of the data structure, and for every $\Delta \in \Xi$, we recursively build a subtree corresponding to the data structure for H_Δ. The space $S(n)$ occupied by this data structure obeys the recursion

$$S(n) \leq O(r^{\lfloor d/2 \rfloor}) + O(r^{\lfloor d/2 \rfloor}) S(n/r),$$

which for a sufficiently large but constant value of r solves to $O(n^{\lfloor d/2 \rfloor + \delta})$.

We now describe a query answering algorithm finding a witness as well. A query with a point q on H is answered as follows: We begin at the root. Being in a non-leaf node v, we determine whether q belongs to some simplex of Ξ_v, the cutting stored at v, and if there is such a simplex, we proceed recursively into the corresponding child of v. If there is no such simplex, it means that there is a hyperplane strictly above q.

In this situation, we shoot a vertical ray ρ from q upwards, we find the first simplex Δ of Ξ_v intersected by ρ and we recurse in the child corresponding to Δ. Some of the hyperplanes relevant for Δ must lie above q: Indeed, let us take the last hyperplane h encountered along ρ in the upward direction. The point $q \cap \rho$ has level 0, and so it has to be contained in some simplex. Therefore Δ contains this point or some point below it, and so h is relevant for Δ, showing the correctness of this step. In a leaf node, we solve the query by inspecting all the hyperplanes in that node. This finishes the query algorithm. Since we spend a constant time in every node, the query time is $O(\log n)$.

The reader familiar with the dynamic data structure of [AM91] can easily verify that the same method works also for that structure and provides a witness hyperplane, while retaining the same ($O(\log n)$) query time. Since the dynamic data structure requires a larger (nonconstant) value of r, we need some auxiliary data structure to find the first simplex hit by the ray ρ. This is easily done by point location in the projection of the cutting Ξ_v on a horizontal hyperplane, which does not affect the space or query time requirements.

Finally we point out that the above described method can be applied also for ray shooting with rays originating above all hyperplanes. In this case, we always recurse in the *last* simplex of the cutting Ξ_v intersected by the query ray; the correctness of this approach is proved similarly as above. In the dynamic data structure with nonconstant r, we again use a point location structure for finding this last simplex. This time we map the problem into a higher dimensional space, where the loci corresponding to rays with the same last simplex induce a subdivision. We conclude that ray shooting of the considered form can be performed with $O(n^{\lfloor d/2 \rfloor + \delta})$ space and preprocessing time, $O(\log n)$ query time and $O(n^{\lfloor d/2 \rfloor - 1 + \delta})$ amortized update time.

The possibility of reducing the space and preprocessing time while retaining a logarithmic query time for the halfspace emptiness problem and the related ray shooting problems was investigated in [Sch92]. One of the solutions obtained there gives $O(n^{\lfloor d/2 \rfloor}(\log n)^{O(1)})$ space and preprocessing time and $O(\log n)$ query time.

2.3 Space/query time tradeoff, further properties of the algorithms

The above described algorithms can be combined, obtaining a continuous tradeoff between space and query time. Such combination is rather standard (see e.g., [CSW90]) and we omit the details. For halfspace emptiness queries, we obtain the following:

Theorem 2.4 *Given an n point set P in E^d ($d \geq 4$) and a parameter m ($n \leq m \leq n^{\lfloor d/2 \rfloor}$), there is a data structure for halfspace emptiness queries with witness with space and preprocessing time $O(m^{1+\delta})$, and with query time $t(n,m) = O(n/m^{1/\lfloor d/2 \rfloor} \log n)$. The structure supports updates with amortized update time $O(m^{1+\delta}/n)$.* □

Also, applying the combined data structures for ray shooting instead of halfspace emptiness queries, we obtain Theorem 1.5.

For the parametric search technique in the sequel, we will also need an efficient parallel version of the query answering algorithms for halfspace emptiness. Actually, for the algorithms with logarithmic query time, a sequential version is good enough for our purposes, so we only need to parallelize the algorithm with linear space (and the combined algorithm). Based on the above description, the reader may check that the linear space algorithm can be parallelized with $O(\log n)$ parallel time and the number of processors proportional to the sequential query time. The combined algorithm with space/query time tradeoff then answers a query in $\tau(n,m) = O(\log n)$ parallel steps using $\pi(n,m) = O(n/m^{1/\lfloor d/2 \rfloor})$ processors.

The subsequent considerations will actually need only little information about the specific halfspace emptiness algorithm used. The running time analysis will be expressed in terms of $t(n,m)$, $\tau(n,m)$ and $\pi(n,m)$ (we only assume that $\tau(n,m) = (\log n)^{O(1)}$ and $\log \pi(n,m) = O(\log n)$). We will however require that all the information the halfspace emptiness algorithm needs about the bounding hyperplane h of the query halfspace γ can be inferred solely from answers to questions of the form "what is the relation of h to a point q (above, or, or below)?" (the point q may depend on P and on the answers to previous questions, but not on h directly). Actually, we will apply the algorithms in the dual setting, where the permitted question is of the form "what is the relation of the query point p to some hyperplane h?". For the above presented halfspace emptiness algorithms one easily verifies that their elementary tests are of this form.

3 Multidimensional parametric search

Let us consider the following problem: We are given n real numbers $x_1, \ldots, x_n$, and for some real number z, we want to decide the order relation (equal, less than, or greater than) between z and each x_i. We do not explicitly know the value of z, but we suppose that an algorithm is available which decides the order relative to z for any given number x. The calls to this algorithm are expensive, so we want to make as few calls as possible.

A well-known solution to this problem is to sort $x_1, \ldots, x_n$, and then use binary search to locate the position of z among these numbers. In this way, we can find a point z' such that the order relation of z' to each of x_i is the same as for z, by $O(\log n)$ calls to the order decision algorithm. All remaining order relations are then determined by direct comparisons with z'.

This is a view of binary search suitable for generalization into higher dimensional setting. Instead of points $x_1, \ldots, x_n \in \mathrm{E}^1$, we consider nonvertical hyperplanes $h_1, \ldots, h_n$ in E^d, and z will a point in E^d. The relation we want to determine is whether z lies below, on, or above h_i, and we again assume that we have an algorithm which can answer such a question for any given hyperplane h. In one step of the generalized binary search, we want to determine the relation of z to a constant fraction of the h_i's using a constant number of calls to the decision algorithm. Such a procedure is contained in Megiddo's paper [Meg84], and a refined version in [Cla86]. One can also use a construction which originated in the context of random sampling methods: For any collection of hyperplanes, the space can be partitioned into a constant number of simplices in such a way than no simplex is intersected by more than (say) half of the hyperplanes. The most economical construction of this type was given in [CF90], and in [Mat91a] it was shown that it can be performed in linear time. Either of these methods can be used to eliminate a constant fraction of hyperplanes. Performing this $O(\log n)$ times, we are left with only a constant number of hyperplanes with an unknown relation to z, which can then be tested directly.

Let us recall Megiddo's parametric search technique, first described in [Meg83] and then applied in the design of numerous algorithms. The basic idea behind his method is as follows: Suppose we have a problem $\mathcal{P}(z)$ that receives as input n data items and a real parameter z. We want to find a value $z^\star$ of z at which the output of $\mathcal{P}(z)$ satisfies certain properties. Suppose we have an efficient sequential algorithm A for solving $\mathcal{P}(z)$ at any given z, and also an algorithm B which can answer questions whether the given z is equal to, less than, or greater than the desired value $z^\star$ (often A and B are actually the same algorithm). Assume moreover that the flow of execution of A depends on comparisons of z with some values, which only depend on the input items and on the outcomes of previous comparisons, but not on z directly.

Megiddo's technique then runs the algorithm A "generically", without specifying the value of z, with the intention of simulating its execution at the unknown $z^\star$. Each time a comparison is to be made, we run algorithm B "off line", thereby determining the outcome of the comparison at $z^\star$. Then execution of the generic A can be resumed. As we proceed through this execution, each question that we resolve further constrains the range where $z^\star$ can lie, and we thus obtain a sequence of progressively smaller intervals, each known to contain $z^\star$, until we reach the end of the generic A with a final interval I. It follows that the outcome of A will be combinatorially the same when we run it on any $z \in I$, including $z^\star$. If I is a singleton, we have found $z^\star$; otherwise it is often straightforward to determine $z^\star$, depending on the nature of the problem — in many cases any value $z^\star \in I$ would do.

The cost of this implicit search is usually dominated by CT_B, where C is the maximum number questions answered by B and T_B is the running time of B. Since this bound is generally too high, Megiddo suggests to replace the generic algorithm A by its parallel version, A_p. If A_p uses π processors and runs in τ parallel steps, then each such step involves at most π *independent* comparisons, that is, each can be carried out without having to know the outcome of the others. At this point, we can apply the binary search strategy to resolve these comparisons. This requires $O(\pi + T_B \log \pi)$ time per parallel step, for a total of $O(\pi\tau + T_B \tau \log \pi)$ time, which often results in a saving of nearly an order of magnitude in the running time. An improvement of this technique by Cole [Col87] can further reduce the running time in certain cases by another logarithmic factor.

In multidimensional parametric search, we assume that the questions asked by the parallel algorithm A_p are not just comparisons of some real numbers with z, but rather they are of the form "what is the relation of z to a hyperplane q", and the algorithm B can answer questions of this type. We then simply replace the usual binary search by the above outlined multidimensional search. This allows us to answer one round of π questions generated by the parallel algorithm by $O(\log \pi)$ calls to the algorithm B, with $O(\pi)$ time overhead.

A similar approach has been used by Cole et al. [CSY87] for a problem in the plane, and in higher dimensions by Cohen and Megiddo [CM89] and independently by Norton et al. [NPT90] (the authors are indebted to N. Megiddo for pointing out the latter two references). However simple, this extension of the para-

metric search technique seems to have many potential applications in computational geometry.

4 Separation queries

We are ready to prove Theorem 1.1. It will be more convenient to work in the dual space (as we already did for the halfspace emptiness queries with logarithmic query time). The collection P of points dualizes to a collection H of nonvertical hyperplanes, and a hyperplane lying completely outside $\mathrm{conv}(P)$ dualizes to a point z completely above or completely below all hyperplanes of H. The condition that the hyperplane should pass through a given point x means in dual that z should be contained in a query hyperplane f. We will restrict ourselves to the "above" case (the "below" case can be treated similarly). We thus have the following query problem for a collection H of hyperplanes: Given a nonvertical hyperplane f, decide if there exists a point of f lying strictly above all hyperplanes of H (let us call such a point *good* for short).

We generalize this query problem to the following problem $\Pi_k(f)$: given a nonvertical k-flat $f \subseteq \mathrm{E}^d$, find a good point in f. We are actually interested in solving $\Pi_{d-1}(f)$, but we will build the solution to all Π_k by induction on k. In order that this induction works, we need the following extended type of solution to $\Pi_k(f)$: If a good point exists, return one, but if it does not, return a $(k+1)$-tuple $\Xi(f) \subseteq H$ such that each point of f lies below some $h \in \Xi(f)$ ($\Xi(f)$ will be called an *obscuring* $(k+1)$*-tuple* for f). The existence of such a $(k+1)$-tuple follows from a dual version of Caratheodory's Theorem on convex sets, and we can certainly find such a $(k+1)$-tuple in time $O(|H|^{k+1})$ by inspecting all $(k+1)$-tuples. We will have a more efficient way to do it, but this observation will be needed for smaller collections of hyperplanes.

The preprocessing in our solution will be the same as for an halfspace emptiness algorithm (one of the algorithms from section 2) or, more precisely, for its dual version, solving the upper envelope problem. Using such a data structure, we will be able to solve all the problems Π_k.

Let us begin by the case $k = 0$. Then f is just a point, and thus problem $\Pi_0(f)$ can be solved in time $t_0(n,m) = t(n,m)$; the obscuring 1-tuple is just a witness hyperplane.

Let now $k > 0$. To solve $\Pi_k(f)$, we will apply k-dimensional parametric search technique, searching for a good point $z^* \in f$. The role of the generic algorithm A will be played by an algorithm deciding whether a point z is good, with a witness in the negative case.

We now need the algorithm B answering questions about the position of z^* within f relative to a given hyperplane q. It will work as follows: Given such a question, we consider the $(k-1)$-flat $f' = f \cap q$ and we solve problem $\Pi_{k-1}(f')$. If this finds a good point $z^* \in f'$, then also the original problem $\Pi_k(f)$ is solved. Otherwise we have an obscuring k-tuple $\Xi = \Xi(f')$. Since the region of points in f *not* obscured by Ξ is convex, it may lie on one side of q only, and thus at least one of the halfspaces determined by f' in f is also obscured by Ξ, and only the other one remains as a potential location of z^*. This suggests an answer about the position of z^* relative to q, which is correct provided that a good point in f exists at all.

When the computation of the generic algorithm is finished without finding a good point in the calls to algorithm B, the possible position of z^* is restricted to an open convex region R, which is an intersection of at most $O(\tau(n,m)\log\pi(n,m))$ of open halfspaces in f (each halfspace corresponding to one call to algorithm B). Let us discuss the various cases arising here.

First suppose that the generic algorithm answers YES (there is a good point). If the region R is nonempty, any of its points must be good (since for any such point, the computation of algorithm A would conclude this). If R is empty, then the answers to the questions must have been inconsistent, and since all the questions are correctly answered provided that a good point exists, the conclusion is that no good point exists. Therefore there must be an obscuring $(k+1)$-tuple, and it can be found among the hyperplanes forming the obscuring k-tuples computed in answering the questions by algorithm B. By testing all $(k+1)$-tuples of these $O(\tau(n,m)\log\pi(n,m))$ hyperplanes, we find that an obscuring $(k+1)$-tuple is found in time $O((\tau(n,m)\log\pi(n,m))^{k+1})$.

Let us now assume that the generic algorithm answers NO; in this case it has to exhibit a witness, which will be one specific hyperplane $h \in H$. Since the computation of algorithm A would be the same for every specific point of the region R, this h has to obscure all points of R. Together with the obscuring k-tuples found in the computations of algorithm B, we again have a small collection of hyperplanes for which we know that it contains an obscuring $(k+1)$-tuple.

The parallel running time of algorithm A is at most $\tau(n,m)$ with $\pi(n,m)$ processors and the running time of algorithm B is $t_{k-1}(n,m)$ by inductive hypothesis. We thus get that the running time of the algorithm solving problem Π_k is

$$O\Big(t_{k-1}(n,m)\tau(n,m)\log\pi(n,m) + \\ + (\tau(n,m)\log\pi(n,m))^{k+1}\Big) = \\ = O(t(n,m)\tau(n,m)^k\log^k\pi(n,m)).$$

With the algorithm for halfspace emptiness queries described in Section 2, we get

$$t_k(n,m) = O(\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^{2k+1} n).$$

(For the case of maximal space $m = n^{\lfloor d/2 \rfloor}$, we have $\pi(n,m) = 1$ and the query time reduces to $O(\log^{k+1} n)$). This finishes the proof of Theorem 1.1. □

Proof of Theorem 1.2: In order to test whether a point x is extreme in a set P, we need a separation query with x on the point set $P \setminus \{x\}$. It is straightforward to extend the halfspace emptiness algorithm from section 2 in such a way that it declares a closed halfspace empty even if its bounding hyperplane constrains only the point x: Such a hyperplane is 0-shallow, hence the fact that x belongs to the bounding hyperplane can be found only in a leaf of the partition tree, so one simply excludes the test involving x in that leaf. A similar modification can be made in the dynamic version of Clarkson's algorithm. We then get an algorithm which can perform a separation query with x on $P \setminus \{x\}$ for any $x \in P$, within the same time bounds as in Theorem 1.1. Performing n such queries finds all extreme points, and the time bound follows by setting $m = n^{2-2/(\lfloor d/2 \rfloor + 1)}$. □

5 Extension to the linear programming case

In this section, we extend the algorithm from the previous sections for handling linear optimization queries. Let us consider a set Γ_0 of n closed halfspaces defining constraints in our linear programming problem. We will assume that the intersection of these halfspaces is nonempty and contains an interior point o (if this is not the case, then either the linear programming problem has no admissible solution, or its dimensionality can be reduced). Using a duality with center o, we get that the admissibility of a point z in a linear programming problem $\mathrm{LP}(\Gamma_0, c)$ translates to the emptiness of a closed halfspace in the dual space. Hence the admissibility can be checked using an algorithm for halfspace emptiness. If a set Γ_q of additional constraints is added to a linear optimization query, the admissibility checking algorithm can obviously be modified to check the admissibility with respect to $\Gamma_0 \cup \Gamma_q$ with $O(|\Gamma_q|)$ increase in the query time; thus adding new constraints with the query presents no problem and we will just consider the minimization problem over an intersection of halfspaces of a set Γ, whose bounding hyperplanes form a set H. For simplicity, we will again assume that the hyperplanes of H are in general position, and that the optimal solution to the considered linear programming problem is unique. With some more care, these assumptions can be removed.

The solution to a linear optimization query will be analogous to the separation query solution in the previous section. This time problem $\Pi_k(f)$ will be the following: For a k-flat $f \subseteq \mathrm{E}^d$, find an admissible point $z^* \in f$ minimizing $c \cdot z$ over the intersection of the halfspaces of Γ with f, and a *defining k-tuple* of this z^*, i.e. a k-tuple of hyperplanes from H whose common intersection with f is the point z^*. If no admissible points exist in f, an obscuring $(k+1)$-tuple $\Xi \subseteq H$ should be output (Ξ witnesses the non-admissibility for all points of f). Our original problem can be solved as soon as we exhibit a solution to $\Pi_d(\mathrm{E}^d)$.

The problem $\Pi_0(f)$ is just the admissibility checking for a point f (requiring a witness halfspace for non-admissibility). To solve $\Pi_k(f)$ for $k > 0$, we run the admissibility checking algorithm generically, and we require an algorithm answering questions about the position of the optimal solution z^* in f relative to given hyperplanes. To answer such a question for a hyperplane q, we solve $\Pi_{k-1}(q \cap f)$. If an optimal solution z' is found, its defining $(k-1)$-tuple guarantees that the value of the objective function in one of the halfspaces defined by q in f cannot exceed the value at z', and thus the other halfspace must be given as the answer. If an obscuring k-tuple is returned, similarly as for the separation queries one of the halfspaces in f is excluded as a possible location of admissible points.

When the generic algorithm finishes its work, the position of the optimal solution z^* is again restricted to an intersection R of a polylogarithmic number of halfspaces in f. If the generic algorithm concludes that an admissible point exists and R is nonempty, we may use a "brute force" linear programming algorithm (going through all points defined by k-tuples of constraints returned as either defining $(k-1)$-tuples or obscuring k-tuples) to optimize the objective function over R and hence compute the solution to the problem $\Pi_k(f)$. If this optimization problem happens to be unbounded, also the original linear programming problem is unbounded and the whole computation may finish. The other cases are similar to the separation query algorithm. Also the running time analysis for this algorithm is the same as for the separation queries. This concludes the proof of Theorem 1.3. □

References

[AM91] P. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. Tech. Report CS–1991–43, Duke University, 1991.

[AM92] P. Agarwal and J. Matoušek. Ray shooting and parametric search. In *24th Symp. on Theory of Computing*, 1992. To appear. Also published as Tech. Report CS-1991-22, Duke University, 1991.

[CF90] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.

[Cla86] K. Clarkson. Linear programming in $O(n \times 3^{d^2})$ time. *Information Processing Letters*, 22(1):21–24, 1986.

[Cla88a] K. Clarkson. Las Vegas algorithm for linear programming when the dimension is small. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 452–457, 1988.

[Cla88b] K. L. Clarkson. Applications of random sampling in computational geometry II. In *Proc. 4th ACM Symposium on Computational Geometry*, pages 1–11, 1988.

[CM89] E. Cohen and N. Megiddo. Strongly polynomial-time and NC algorithms for detecting cycles in dynamic graphs. In *Proc. 21st ACM Symposium on Theory of Computing*, pages 523–534, 1989.

[Col87] R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *Journal of the ACM*, 34:200–208, 1987.

[CSW90] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. In *Proc. 6. ACM Symposium on Computational Geometry*, pages 23–33, 1990.

[CSY87] R. Cole, M. Sharir, and C. Yap. On k-hulls and related problems. *SIAM Journal on Computing*, 16(1):61–67, 1987.

[Dye86] E. M. Dyer. On a multidimensional search technique and its application to the Euclidean 1-centre problem. *SIAM Journal on Computing*, 15:725–738, 1986.

[Ede87] H. Edelsbrunner. *Algorithms in combinatorial geometry*. Springer, 1987.

[Epp91] D. Eppstein. Dynamic three-dimensional linear programming. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 94–103, 1991.

[GSC87] L. Guibas, J. Stolfi, and K. Clarkson. Solving related two- and three-dimensional linear programming in logarithmic time. *Theoretical Computer Science*, 49(1):81–84, 1987.

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

[Mat91a] J. Matoušek. Approximations and optimal geometric divide-and-conquer. In *Proc. 23nd ACM Symposium on Theory of Computing*, pages 505–511, 1991.

[Mat91b] J. Matoušek. Efficient Partition Trees. In *Proc. 7th ACM Symposium on Computational Geometry*, pages 1–9, 1991.

[Mat91c] J. Matoušek. Reporting points in halfspaces. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 207–215, 1991.

[Mat91d] J. Matoušek. Linear optimization queries. J. Algorithms, to appear.

[MS91] J. Matoušek and O. Schwarzkopf. Linear optimization queries. Tech. Report B 91–19, Freie Universität Berlin, 1991.

[Meg83] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *Journal of the ACM*, 30:852–865, 1983.

[Meg84] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM*, 31:114–127, 1984.

[Meh85] K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*. Springer-Verlag, Heidelberg, 1985.

[Mul91] K. Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, 1991, pages 180–196.

[NPT90] C. H. Norton, S. A. Plotkin and E. Tardos. Using separation algorithms in fixed dimension. In: *Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990, pages 377–378.

[PH77] F. P. Preparata and S. J. Hong. Convex hulls of finite point sets in two and three dimensions. *Communications of the ACM*, 20:89–73, 1977.

[Sch91] O. Schwarzkopf. On the Post Office Problem. Manuscript, 1991.

[Sch92] O. Schwarzkopf. Ray shooting in convex polytopes. This proceedings.

[Sei86] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proc. 18th ACM Symposium on Theory of Computing*, pages 404–413, 1986.

[Sei90] R. Seidel. Low dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6:423–434, 1991.

[SW90] R. Seidel and E. Welzl. Private communication, May 1990.