CMSC 754: Lecture 7
Linear Programming

Reading: Chapter 4 in the 4M's. The original algorithm was given in R. Seidel. Small-dimensional linear programming and convex hulls made easy, *Discrete and Computational Geometry*, vol 6, 423–434, 1991.

Linear Programming: One of the most important computational problems in science and engineering is *linear programming*, or *LP* for short. LP is perhaps the simplest and best known example of multi-dimensional constrained optimization problems. In constrained optimization, the objective is to find a point in d-dimensional space that minimizes (or maximizes) a given *objective function*, subject to satisfying a set of *constraints* on the set of allowable solutions. LP is distinguished by the fact that both the constraints and objective function are *linear functions*. In spite of this apparent limitation, linear programming is a very powerful way of modeling optimization problems. Typically, linear programming is performed in spaces of very high dimension (hundreds to thousands or more). There are, however, a number of useful (and even surprising) applications of linear programming in low-dimensional spaces.

Formally, in *linear programming* we are given a set of linear inequalities, called *constraints*, in real d-dimensional space $\mathbb{R}^d$. Given a point $(x_1, \ldots, x_d) \in \mathbb{R}^d$, we can express such a constraint as $a_1x_1 + \ldots + a_dx_d \leq b$, by specifying the coefficient a_i and b. (Note that there is no loss of generality in assuming that the inequality relation is $\leq$, since we can convert a $\geq$ relation to this form by simply negating the coefficients on both sides.) Geometrically, each constraint defines a closed halfspace in $\mathbb{R}^d$. The intersection of these halfspaces intersection defines a (possibly empty or possibly unbounded) polyhedron in $\mathbb{R}^d$, called the *feasible polytope*[1] (see Fig. 1(a)).

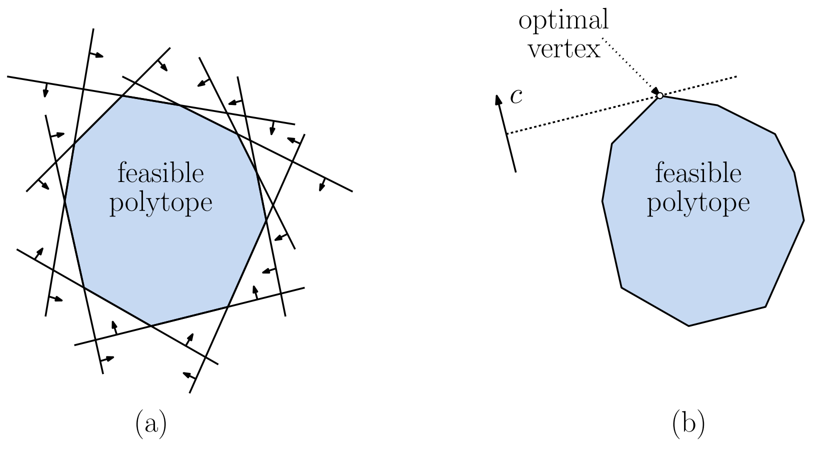

Fig. 1: 2-dimensional linear programming.

We are also given a linear *objective function*, which is to be minimized or maximized subject to the given constraints. We can express such as function as $c_1x_1 + \ldots + c_dx_d$, by specifying the coefficients c_i. (Again, there is no essential difference between minimization and

[1]To some geometric purists this an abuse of terminology, since a polytope is often defined to be a closed, bounded convex polyhedron, and feasible polyhedra need not be bounded.

maximization, since we can simply negate the coefficients to simulate the other.) We will assume that the objective is to maximize the objective function. If we think of $(c_1, \ldots, c_d)$ as a vector in $\mathbb{R}^d$, the value of the objective function is just the projected length of the vector $(x_1, \ldots, x_d)$ onto the direction defined by the vector c. It is not hard to see that (assuming general position), if a solution exists, it will be achieved by a vertex of the feasible polytope, called the *optimal vertex* (see Fig. 1(b)).

In general, a d-dimensional linear programming problem can be expressed as:

$$\begin{array}{ll} \text{Maximize:} & c_1x_1 + c_2x_2 + \cdots + c_dx_d \\ \text{Subject to:} & a_{1,1}x_1 + \cdots + a_{1,d}x_d \leq b_1 \\ & a_{2,1}x_1 + \cdots + a_{2,d}x_d \leq b_2 \\ & \vdots \\ & a_{n,1}x_1 + \cdots + a_{n,d}x_d \leq b_n, \end{array}$$

where $a_{i,j}$, c_i, and b_i are given real numbers. This can be also be expressed in matrix notation:

$$\begin{array}{ll} \text{Maximize:} & c^{\mathsf{T}}x, \\ \text{Subject to:} & Ax \leq b. \end{array}$$

where c and x are d-vectors, b is an n-vector and A is an $n \times d$ matrix. Note that c should be a nonzero vector, and n should be at least as large as d and may generally be much larger.

There are three possible outcomes of a given LP problem:

Feasible: The optimal point exists (and assuming general position) is a unique vertex of the feasible polytope (see Fig. 2(a)).

Infeasible: The feasible polytope is empty, and there is no solution (see Fig. 2(b)).

Unbounded: The feasible polytope is unbounded in the direction of the objective function, and so no finite optimal solution exists (see Fig. 2(c)).

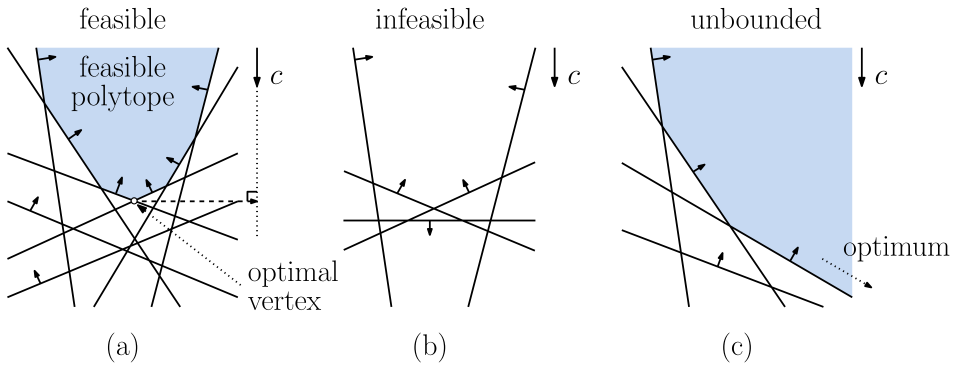

Fig. 2: Possible outcomes of linear programming.

In our figures (in case we don't provide arrows), we will assume the feasible polytope is the intersection of upper halfspaces. Also, we will usually take the objective vector c to be a vertical vector pointing downwards. (There is no loss of generality here, because we can always rotate space so that c is parallel any direction we like.) In this setting, the problem is just that of finding the lowest vertex (minimum y-coordinate) of the feasible polytope.

Linear Programming in High Dimensional Spaces: As mentioned earlier, typical instances of linear programming may involve hundreds to thousands of constraints in very high dimensional space. It can be proved that the combinatorial complexity (total number of faces of all dimensions) of a polytope defined by n halfspaces can be as high as $\Omega(n^{\lfloor d/2 \rfloor})$. In particular, the number of vertices alone might be this high. Therefore, building a representation of the entire feasible polytope is not an efficient approach (except perhaps in the plane).

The principal methods used for solving high-dimensional linear programming problems are the *simplex algorithm* and various *interior-point methods*. The simplex algorithm works by finding a vertex on the feasible polytope, then walking edge by edge downwards until reaching a local minimum. (By convexity, any local minimum is the global minimum.) It has been long known that there are instances where the simplex algorithm runs in exponential time, but in practice it is quite efficient.

The question of whether linear programming is even solvable in polynomial time was unknown until Khachiyan's ellipsoid algorithm (late 70's) and Karmarkar's more practical interior-point algorithm (mid 80's). Both algorithms are polynomial in the total number of bits needed to describe the input. This is called a *weakly polynomial time* algorithm. It is not known whether there is a strongly polynomial time algorithm, that is, one whose running time is polynomial in both n and d, irrespective of the number of bits used for the input coefficients. Indeed, like P versus NP, this is recognized by some as one of the great unsolved problems of mathematics.

Solving LP in Spaces of Constant Dimension: There are a number of interesting optimization problems that can be posed as a low-dimensional linear programming problem. This means that the number of variables (the x_i's) is constant, but the number of constraints n may be arbitrarily large.

The algorithms that we will discuss for linear programming are based on a simple method called *incremental construction*. Incremental construction is among the most common design techniques in computational geometry, and this is another important reason for studying the linear programming problem.

(Deterministic) Incremental Algorithm: Recall our geometric formulation of the LP problem. We are given n halfspaces $\{h_1, \ldots, h_d\}$ in $\mathbb{R}^d$ and an objective vector c, and we wish to compute the vertex of the feasible polytope that is most extreme in direction c. Our incremental approach will be based on starting with an initial solution to the LP problem for a small set of constraints, and then we will successively add one new constraint and update the solution.

In order to get the process started, we need to assume (1) that the LP is bounded and (2) we can find a set of d halfspaces that provide us with an initial feasible point. Getting to this starting point is actually not trivial.[2] For the sake of focusing on the main elements of the algorithm, we will skip this part and just assume that the first d halfspaces define a bounded feasible polytope (actually it will be a polyhedral cone). The the unique point where all d bounding hyperplanes, $h_1, \ldots, h_d$, intersect will be our initial feasible solution. We denote this vertex as v_d (see Fig. 3(a)).

We will then add halfspaces one by one, $h_{d+1}, h_{d+2}, \ldots$, and with each addition we update the current optimum vertex, if necessary. Let v_i denote the optimal feasible vertex after

[2]Our textbook explains how to overcome these assumptions in $O(n)$ additional time.

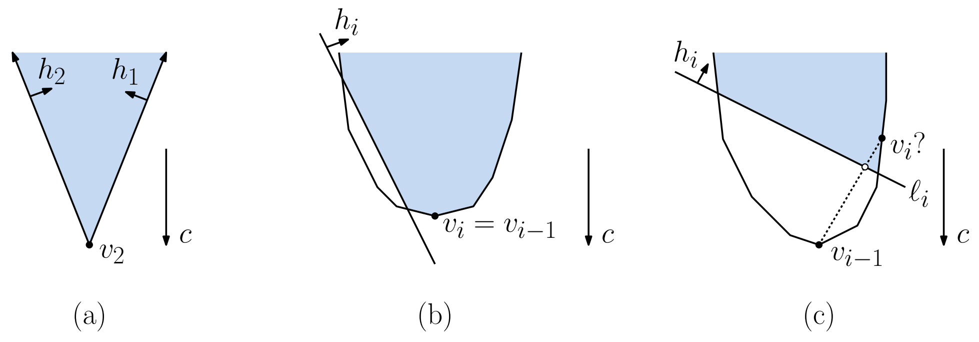

Fig. 3: (a) Starting the incremental construction and (b) the proof that the new optimum lies on ℓ_i.

the addition of $\{h_1, h_2, \ldots, h_i\}$. Notice that with each new constraint, the feasible polytope generally becomes smaller, and hence the value of the objective function at optimum vertex can only decrease. (In terms of our illustrations, the y-coordinate of the feasible vertex increases.)

There are two cases that can arise when h_i is added. In the first case, v_{i-1} lies within the halfspace h_i, and so it already satisfies this constraint (see Fig. 3(b)). If so, then it is easy to see that the optimum vertex does not change, that is $v_i = v_{i-1}$.

In the second case v_{i-1} violates constraint h_i. In this case we need to find a new optimum vertex (see Fig. 4(c)). Let us consider this case in greater detail. The key observation is presented in the following claim, which states that whenever the old optimum vertex is infeasible, the new optimum vertex lies on the bounding hyperplane of the new constraint.

Lemma: If after the addition of constraint h_i the LP is still feasible but the optimum vertex changes, then the new optimum vertex lies on the hyperplane bounding h_i.

Proof: Let ℓ_i denote the bounding hyperplane for h_i. Let v_{i-1} denote the old optimum vertex. Suppose towards contradiction that the new optimum vertex v_i does not lie on ℓ_i (see Fig. 3(c)). Consider the directed line segment $\overline{v_{i-1}v_i}$. Observe first that as you travel along this segment the value of the objective function decreases monotonically. (This follows from the linearity of the objective function and the fact that v_{i-1} is no longer feasible.) Also observe that, because it connects a point that is infeasible (lying below ℓ_i) to one that is feasible (lying strictly above ℓ_i), this segment must cross ℓ_i. Thus, the objective function is maximized at the crossing point itself, which lies on ℓ_i, a contradiction.

Recursively Updating the Optimum Vertex: Using this observation, we can reduce the problem of finding the new optimum vertex to an LP problem in one lower dimension. Let us consider an instance where the old optimum vertex v_{i-1} does not lie within h_i (see Fig. 4(a)). Let ℓ_i denote the hyperplane bounding h_i. We first project the objective vector c onto ℓ_i, letting c' be the resulting vector (see Fig. 4(b)). Next, intersect each of the halfspaces $\{h_1, \ldots, h_{i-1}\}$ with ℓ_i. Each intersection is a $(d-1)$-dimensional halfspace that lies on ℓ_i. Since ℓ_i is a $(d-1)$-dimensional hyperplane, we can project ℓ_i onto $\mathbb{R}^{d-1}$ space (see Fig. 4(b)).

We will not discuss how this is done, but the process is a minor modification of Gauss elimination in linear algebra. We now have an instance of LP in $\mathbb{R}^{d-1}$ involving $i-1$ constraints. We recursively solve this LP. The resulting optimum vertex v_i is then projected back onto ℓ_i and can now be viewed as a point in d-dimensional space. This is the new optimum point that we desire.

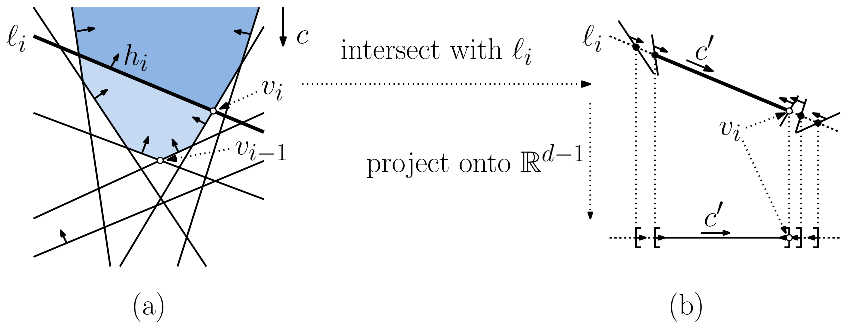

Fig. 4: Incremental construction.

The recursion ends when we drop down to an LP in 1-dimensional space (see Fig. 4(b)). The projected objective vector c' is a vector pointing one way or the other on the real line. The intersection of each halfspace with ℓ_i is a ray, which can be thought of as an interval on the line that is bounded on one side and unbounded on the other. Computing the intersection of a collection of intervals on a line can be done easily in linear time, that is, $O(i-1)$ time in this case. (This interval is the heavy solid line in Fig. 4(b).) The new optimum is whichever endpoint of this interval is extreme in the direction of c'. If the interval is empty, then the feasible polytope is also empty, and we may terminate the algorithm immediately and report that there is no solution. Because, by assumption, the original LP is bounded, it follows that the $(d-1)$-dimensional LP is also bounded.

Worst-Case Analysis: What is the running time of this algorithm? Ignoring the initial d halfspaces, there are $n-d$ halfspace insertions performed. In step i, we may find that the current optimum vertex is feasible. This takes $O(d)$ time. The alternative is that we need to solve a $(d-1)$-dimensional LP with $i-1$ constraints. It takes $O(d(i-1))$ to intersect each of the constraints with ℓ_i and $O(d)$ time to project c onto ℓ_i. If we let $T_d(n)$ denote the time to run this algorithm in dimension d with n constraints. In this case the time is $O(di+T_{d-1}(i-1))$. Since there are two alternatives, the running time is the maximum of the two. Ignoring constant factors, the running time can be expressed by the following recurrence formula:

$$T_d(n) \;=\; \sum_{i=d+1}^{n} \max\big(d, di+T_{d-1}(i-1)\big).$$

Since d is a constant, we can simplify this to:

$$T_d(n) \;=\; \sum_{i=d+1}^{n} \big(i+T_{d-1}(i-1)\big).$$

The basis case of the recurrence occurs when $d = 1$, and we just solve the interval intersection problem described above in $O(n)$ time by brute force. Thus, we have $T_1(n) = n$. It is easy to verify by induction [3] that this recurrence solves to $T_d(n) = O(n^d)$, which is not very efficient.

Notice that this worst-case analysis is based on the rather pessimistic assumption that the current vertex is *always infeasible*. Although there may exist insertion orders for which this might happen, we might wonder whether we can arrange the insertion order so this worst case does not occur. We'll consider this alternative next.

Randomized Algorithm: Suppose that we apply the above algorithm, but we insert the halfspaces in *random order* (except for the first d, which need to be chosen to provide an initial feasible vertex.) This is an example of a general class of algorithms called *randomized incremental algorithms*. A description is given in the code block below.

Randomized Incremental d-Dimensional Linear Programming

Input: A set $H = \{h_1, \ldots, h_n\}$ of $(d-1)$-dimensional halfspaces, such that the first d define an initial feasible vertex v_d, and the objective vector c.
Output: The optimum vertex v or an error status indicating that the LP is infeasible.

```
(1) If the dimension is 1, solve the LP by brute force in $O(n)$ time.
(2) Let $v_d$ be the intersection point of the hyperplanes bounding $h_1, \ldots, h_d$, which we assume define an
    initial feasible vertex. Randomly permute the remaining halfspaces, and let $\langle h_{d+1}, \ldots, h_n \rangle$ denote the
    resulting sequence.
(3) For $i = d+1$ to $n$ do:
    (a) If $(v_{i-1} \in h_i)$ then $v_i \leftarrow v_{i-1}$.
    (b) Otherwise, intersect $\{h_1, \ldots, h_{i-1}\}$ with the $(d-1)$-dimensional hyperplane $\ell_i$ that bounds $h_i$ and
        project onto $\mathbb{R}^{d-1}$. Let $c'$ be the projection of $c$ onto $\ell_i$ and then onto $\mathbb{R}^{d-1}$. Solve the resulting
        $(d-1)$-dimensional LP recursively.
        (i) If the $(d-1)$-dimensional LP is infeasible, terminate and report that the LP is infeasible.
        (ii) Otherwise, let $v_i$ be the solution to the $(d-1)$-dimensional LP.
(4) Return $v_n$ as the final solution.
```

What is the expected case running time of this randomized incremental algorithm? Note that the expectation is over the random permutation of the insertion order. We make *no assumptions* about the distribution of the input. (Thus, the analysis is in the worst-case with respect to the input, but in the expected case with respect to random choices.)

The number of random permutations is $(n-d)!$, but it will simplify things to pretend that we permute all the halfspaces, and so there are $n!$ permutations. Each permutation has an equal probability of $1/n!$ of occurring, and an associated running time. However, presenting the analysis as sum of $n!$ terms does not lead to something that we can easily simplify. We will apply a technique called *backwards analysis*, which is quite useful.

[3] Suppose inductively that there exists a sufficiently large constant α such that $T_d(n) \le \alpha n^d$. The basis case is trivial. Assuming the induction hypothesis holds for dimension $d-1$, we have

$$T_d(n) \;=\; \sum_{i=d+1}^{n} \big(i + T_{d-1}(i-1)\big) \;\le\; \sum_{i=d+1}^{n} \big(i + \alpha(i-1)^{d-1}\big) \;\le\; \sum_{i=1}^{n} \alpha n^{d-1} \;\le\; \alpha n^d.$$

Although this analysis is quite crude, it can be shown to be asymptotically tight.

Computing the Minimum (Optional): To motivate how backwards analysis works, let us consider a much simpler example, namely the problem of computing the minimum. Suppose that we are given a set S of n distinct numbers. We permute the numbers and inspect them one-by-one. We maintain a variable that holds the smallest value seen so far. If we see a value that is smaller than the current minimum, then we *update* the current smallest. Of course, this takes $O(n)$ time, but the question we will consider is, in expectation *how many times does the current smallest value change*?

Below are three sequences that illustrate that the minimum may updated once (if the numbers are given in increasing order), n times (if given in decreasing order). Observe that in the third sequence, which is random, the minimum does not change very often at all.

$\underline{0}$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\underline{14}$	$\underline{13}$	$\underline{12}$	$\underline{11}$	$\underline{10}$	$\underline{9}$	$\underline{8}$	$\underline{7}$	$\underline{6}$	$\underline{5}$	$\underline{4}$	$\underline{3}$	$\underline{2}$	$\underline{1}$	$\underline{0}$
$\underline{5}$	9	$\underline{4}$	11	$\underline{2}$	6	8	14	$\underline{0}$	3	13	12	1	7	10

Let p_i denote the probability that the minimum value changes on inspecting the ith number of the random permutation. Thus, with probability p_i the minimum changes (and we add 1 to the counter for the number of changes) and with probability $1 - p_i$ it does not (and we add 0 to the counter for the number of changes). The total expected number of changes is

$$C(n) \;=\; \sum_{i=1}^{n}(p_i \cdot 1 + (1-p_i)\cdot 0) \;=\; \sum_{i=1}^{n} p_i.$$

It suffices to compute p_i. We might be tempted to reason as follows. Let us consider a random subset of the first $i-1$ values, and then consider all the possible choices for the ith value from the remaining $n-i+1$ elements of S. However, this leads to a complicated analysis involving conditional probabilities. (For example, if the minimum is among the first $i-1$ elements, $p_i = 0$, but if not then it is surely positive.) Let us instead consider an alternative approach, in which we work *backwards*. In particular, let us fix the first i values, and then consider the probability the *last value added to this set resulted in a change in the minimum.*

To make this more formal, let S_i be an arbitrary subset of i numbers from our initial set of n. (In theory, the probability is conditional on the fact that the elements of S_i represent the first i elements to be chosen, but since the analysis will not depend on the particular choice of S_i, it follows that the probability that we compute will hold unconditionally.) Among all the $n!$ permutations that could have resulted in S_i, each of the $i!$ permutations of these first i elements are equally likely to occur. For how many of these permutations does the minimum change in the transition from S_{i-1} to S_i? Clearly, the minimum changes only for those sequences in which the smallest element of S_i is the ith element itself. Since the minimum item appears with equal probability in each of the i positions of a random sequence, the probability that it appears last is exactly $1/i$. Thus, $p_i = 1/i$. From this we have

$$C(n) \;=\; \sum_{i=1}^{n} p_i \;=\; \sum_{i=1}^{n} \frac{1}{i} \;=\; \ln n + O(1).$$

This summation $\sum_i \frac{1}{i}$ is the *Harmonic series*, and it is a well-known fact that it is nearly equal to $\ln n$. (See any text on probability theory.)

Note that by fixing S_i, and considering the possible (random) transitions that lead from S_{i-1} to S_i, we avoided the need to consider any conditional probabilities. This is called a *backwards analysis* because the analysis works by considering the possible random transitions that brought us to S_i from S_{i-1}, as opposed to working forward from S_{i-1} to S_i. Of course, the probabilities are no different whether we consider the random sequence backwards rather than forwards, so this is a perfectly accurate analysis. It's arguably simpler and easier to understand.

Backwards Analysis for Randomized LP: Let us apply this same approach to the analysis of the running time of the randomized incremental linear programming algorithm. We will do the analysis in d-dimensional space. Let $T_d(n)$ denote the expected running time of the algorithm on a set of n halfspaces in dimension d. We will prove by induction that $T_d(n) \leq \gamma\, d!\, n$, where γ is some constant that does not depend on dimension. It will make the proof simpler if we start by proving that $T_d(n) \leq \gamma_d\, d!\, n$, where γ_d does depend on dimension, and later we will eliminate this dependence.

For $d + 1 \leq i \leq n$, let p_i denote the probability that the insertion of the ith hyperplane in the random order results in a change in the optimum vertex.

Case 1: With probability $(1 - p_i)$ there is no change. It takes us $O(d)$ time to determine that this is the case.

Case 2: With probability p_i, there is a change to the optimum. First we project the objective vector onto ℓ_i (which takes $O(d)$ time), next we intersect the existing $i - 1$ halfspaces with ℓ_i (which takes $O(d(i - 1))$ time). Together, these last two steps take $O(di)$ time. Finally we invoke a $(d - 1)$-dimensional LP on a set of $i - 1$ halfspaces in dimension $d-1$. By the induction hypothesis, the running time of this recursive call is $T_{d-1}(i-1)$.

Combining the two cases, up to constant factors (which don't depend on dimension), we have a total expected running time of

$$T_d(n) \;\leq\; \sum_{i=d+1}^{n} \Big((1-p_i)d + p_i\big(di + T_{d-1}(i)\big)\Big) \;\leq\; \sum_{i=d+1}^{n} \big(d + p_i\big(di + T_{d-1}(i)\big)\big).$$

It remains is to determine what p_i is. To do this, we will apply the same backward-analysis technique as above. Let S_i denote an arbitrary subset consisting of i of the original halfspaces. Again, it will simplify things to assume that all the i hyperplanes are being permuted (not just the last $i - d$). Among all $i!$ permutations of S_i, in how many does the optimum vertex change with the ith step? Let v_i denote the optimum vertex for these i halfspaces. It is important to note that v_i depends only on the set S_i and not on the order of their insertion. (You might think about why this is important.)

Assuming general position, there are d halfspaces whose intersection defines v_i. (For example, in Fig. 5(a), we label these halfspaces as h_4 and h_7.)

- If none of these d halfspaces were the last to be inserted, then $v_i = v_{i-1}$, and there is no change. (As is the case in Fig. 5(b), where h_5 is the last to be inserted.)

- On the other hand, if any of them were the last to be inserted, then v_i did not exist yet, and hence the optimum must have changed as a result of this insertion. (As is the case in Fig. 5(c), where h_7 is the last to be inserted.)

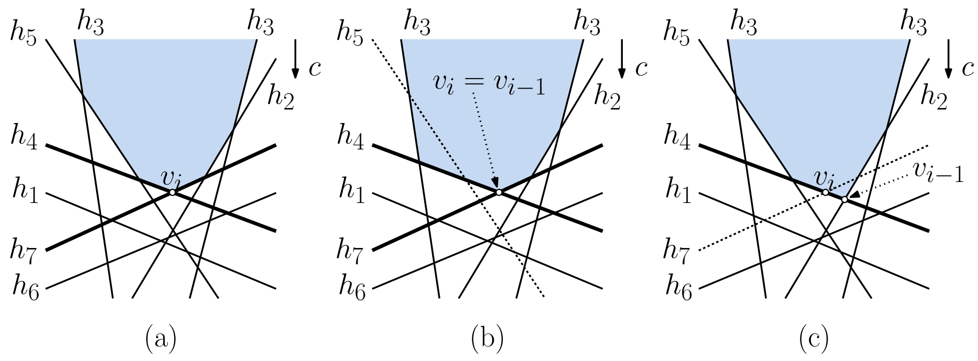

Fig. 5: Backwards analysis for the randomized LP algorithm.

Thus, the optimum changes if and only if either one of the d defining halfspaces was the last halfspace inserted. Since all of the i halfspaces are equally likely to be last, this happens with probability d/i. Therefore, $p_i = d/i$.

This probabilistic analysis has been conditioned on the assumption that S_i was the subset of halfspace seen so far, but since the final probability does not depend on any properties of S_i (just on d and i), the probabilistic analysis applies unconditionally to all subsets of size i.

Returning to our analysis, since $p_i = d/i$, and applying the induction hypothesis that $T_{d-1}(i) = \gamma_{d-1}(d-1)!\, i$, we have

$$\begin{aligned} T_d(n) &\leq \sum_{i=d+1}^{n} \big(d + p_i\big(di + T_{d-1}(i)\big)\big) \;\leq\; \sum_{i=d+1}^{n} \left(d + \frac{d}{i}\big(di + \gamma_{d-1}(d-1)!\, i\big)\right) \\ &\leq \sum_{i=d+1}^{n} (d + d^2 + \gamma_{d-1} d!) \;\leq\; (d + d^2 + \gamma_{d-1} d!) n. \end{aligned}$$

To complete the proof, we just need to select γ_d so that the right hand side is at most $\gamma_d d!$. To achieve this, it suffices to set

$$\gamma_d \;=\; \frac{d + d^2}{d!} + \gamma_{d-1}.$$

Plugging this value into the above formula yields

$$T_d(n) \;\leq\; (d + d^2 + \gamma_{d-1} d!) n \;\leq\; \left(\frac{d + d^2}{d!} + \gamma_{d-1}\right) d!\, n \;\leq\; \gamma_d d!\, n,$$

as desired.

Eliminating the Dependence on Dimension: As mentioned above, we don't like the fact that the "constant" γ_d changes with the dimension. To remedy this, note that because $d!$ grows so rapidly compared to either d or d^2, it is easy to show that $(d + d^2)/d! \leq 1/2^d$ for almost all sufficiently large values of d. Because the geometric series $\sum_{d=1}^{\infty} 1/2^d$, converges, it follows that there is a constant γ (independent of dimension) such that $\gamma_d \leq \gamma$ for all d. Thus, we have that $T_d(n) \leq O(d!\, n)$, where the constant factor hidden in the big-Oh does not depend on dimension.

Concluding Remarks: In summary, we have presented a simple and elegant randomized incremental algorithm for solving linear programming problems. The algorithm runs in $O(n)$ time in expectation. (Remember that expectation does *not* depend on the input, only on the random choices.) Unfortunately, our assumption that the dimension d is a constant is crucial. The factor $d!$ grows so rapidly (and it seems to be an unavoidable part of the analysis) that this algorithm is limited to fairly low dimensional spaces.

You might be disturbed by the fact that the algorithm is not deterministic, and that we have only bounded the expected case running time. Might it not be the case that the algorithm takes ridiculously long, degenerating to the $O(n^d)$ running time, on very rare occasions? The answer is, of course, yes. In his original paper, Seidel proves that the probability that the algorithm exceeds its running time by a factor b is $O((1/c)^{b\,d!})$, for any fixed constant c. For example, he shows that in 2-dimensional space, the probability that the algorithm takes more than 10 times longer than its expected time is at most 0.0000000000065. You would have a much higher probability be being struck by lightning *twice* in your lifetime!