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.

Linear programming is perhaps the simplest example of a constrained optimization problem, since 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.

A Bit of History: The problem of solving a system of linear inequalities dates back to at least the 1800's, where it was studied by Joseph Fourier (of "Fourier Transform" fame). Serious study of linear programming started in the 1940's where it was developed (independently) by Soviet mathematician and economist Leonid Kantorovich and George Dantzig and John von Neumann. Dantzig developed the *simplex algorithm*, which provided a practical method for solving large[1] instances of LP.

While simplex is fast in practice, it is known that it can take exponential time. In his original list of NP-hard problems, Richard Karp listed LP as one of the major open problems that are not known to be solvable in polynomial time or NP-hard. In 1979, Leonid Khachiyan had a major breakthrough by showing that LP could be solved in (weakly) polynomial time through a method called the *ellipsoid algorithm.* The term "weakly" means that the running time is polynomial not only in the input size, but also in the number of bits in the numbers involved. The ellipsoid algorithm was primarily of theoretical interest, but in 1984 Narendra Karmarkar developed a class of (also weakly polynomial) algorithms called *interior-point methods.* These are quite practical, and are widely used today.

The question of whether there is a (strongly) polynomial time algorithm for LP is among the most important open problems in computer science (right up there with $P = NP$). In this lecture, we will discuss a linear time algorithm for LP, but with the constraint that the dimension is a fixed constant.

Problem Definition: 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

[1]Well, back in the 1940's, solving 100 inequalities was considered "large"! But today, this algorithm solves instances involving hundreds of thousands constraints.

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*[2] (see Fig. 1(a)).

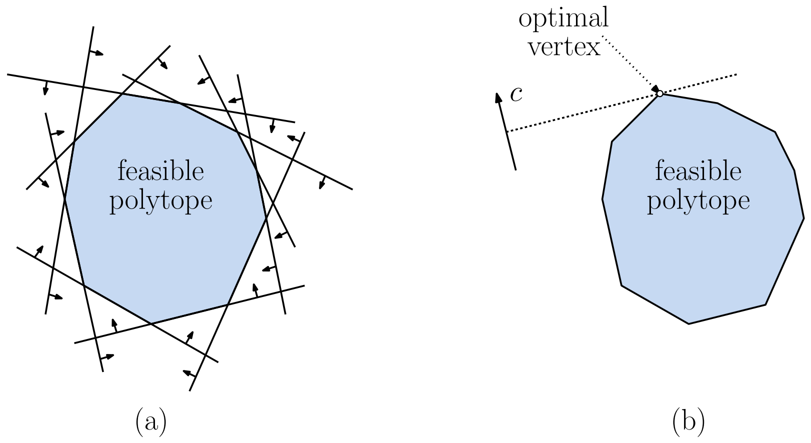

Fig. 1: 2-dimensional linear programming.

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 $c_1x_1 + \cdots + c_dx_d$, by specifying the coefficients c_i. (Again, there is no essential difference between minimization and 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:

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

[2]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.

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)).

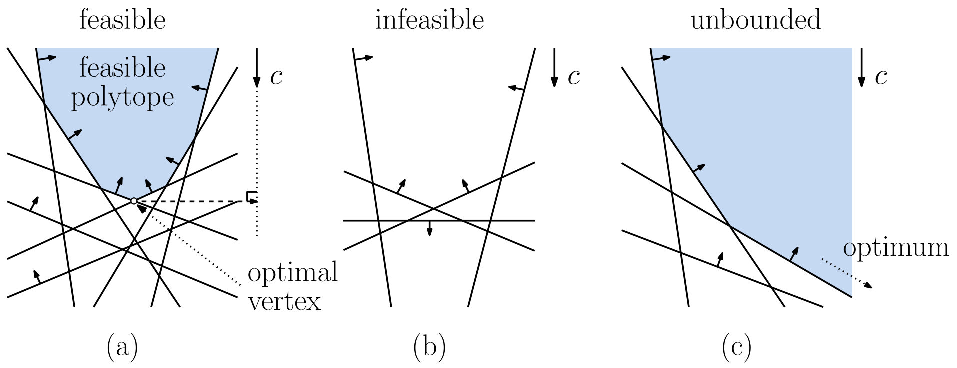

Fig. 2: Possible outcomes of linear programming.

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.

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 $H = \{h_1, \ldots, h_n\}$ 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.[3] 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, $h_{d+1}, h_{d+2}, \ldots, h_n$, and with each addition we update the current optimum vertex, if necessary. For $d \leq i \leq n$, let $H_i = \{h_1, \ldots, h_i\}$ denote the first i constraints, and let v_i denote the associated optimum vertex. Clearly, v_i is the final solution to the LP. (In our figures, c points downwards, so successive v_i's will move upwards as more constraints are considered.)

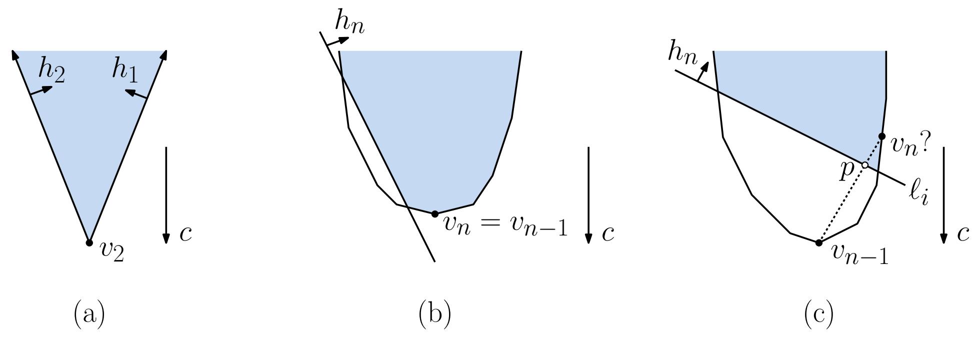

Fig. 3: (a) Starting the incremental construction and (b) the proof that the new optimum lies on ℓ_n.

Let's do this by induction. Assume that we have correctly computed v_{n-1}, based on the first $n-1$ constraints H_{n-1}, and all that remains is to add the final constraint h_n. There are two cases that can arise:

v_{n-1} **is feasible for** h_n**:** ($v_{n-1} \in h_n$. See Fig. 3(b)): The optimum vertex does not change. Set $v_n \leftarrow v_{n-1}$.

v_{n-1} **violates** h_n**:** ($v_{n-1} \notin h_n$. See Fig. 3(c)): We need to update v_n. This is described below.

The key observation for updating the optimum is presented in the following claim, which states that the new optimum vertex lies on the boundary of the new constraint.

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

Proof: Let ℓ_n denote the bounding hyperplane for h_n. Let v_{n-1} denote the old optimum vertex. Suppose towards contradiction that the new optimum vertex v_n does not lie on ℓ_n (see Fig. 3(c)). Since $v_{n-1} \notin h_n$ and $v_n \in h_n$, the line segment $\overline{v_{n-1}v_n}$ crosses ℓ_n. Let $p = \overline{v_{n-1}v_n} \cap \ell_n$. By convexity p is feasible. By linearity the objective function grows progressively worse as we walk from v_{n-1} to v_n. Therefore p is a better solution that v_n, a contradiction.

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

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_{n-1} does not lie within h_n (see Fig. 4(a)). Let ℓ_n denote the bounding hyperplane for the halfspace h_n. We first project the objective vector c onto ℓ_n, letting c' be the resulting vector (see Fig. 4(b)). Next, intersect each of the halfspaces $\{h_1, \ldots, h_{n-1}\}$ with ℓ_n. Each intersection is a $(d-1)$-dimensional halfspace that lies on ℓ_n. Since ℓ_n is a $(d-1)$-dimensional hyperplane, we can project ℓ_n onto $\mathbb{R}^{d-1}$ space (see Fig. 4(b)).

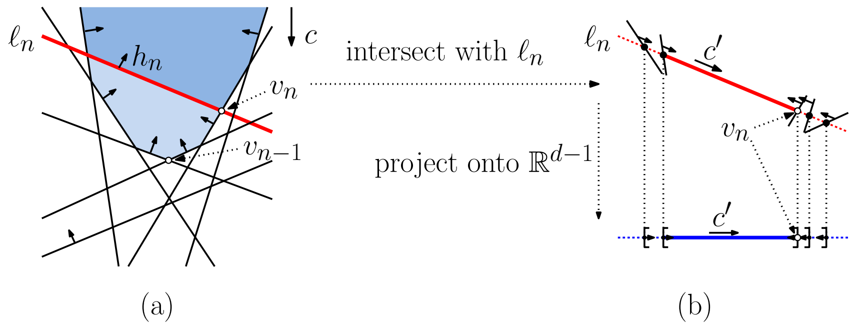

Fig. 4: Incremental construction.

How do we carry this out? We will not discuss this here except to say that this basically involves Gauss elimination, which you probably learned in your course on linear algebra. (See the end of this lecture for details.) All we need to know for now is that this can be done in time $O(dn)$, and results in an LP involving $n-1$ constraints in $d-1$ dimensional space. Once we solve this LP, we can invert the projection process to project the optimum vertex back into $\mathbb{R}^d$, which becomes the new optimum v_n.

The recursion "bottoms out" when we reach 1-dimensional space (see the bottom of Fig. 4(b)). Each constraint is a simple inequality (either $v \leq a_i$ or $v \geq a_i$), and the objective function degenerates to simply "find the largest" or "find the smallest" feasible point. This is trivially solvable in $O(n)$ time. Since the original LP was bounded, the resulting 1-dimensional LP will also be bounded, but it might be infeasible.

Worst-Case Analysis: What is the running time of this algorithm? It turns out that the running time is sensitive to the order in which the halfspaces are inserted. (The complete algorithm is described below.) Let's consider what happens assuming the worst possible insertion order.

Since the algorithm is recursive, it is natural to express the running time as a recurrence. Let $W_d(n)$ denote the worst-case running time of our LP algorithm for n halfspaces in $\mathbb{R}^d$. Recall that we needed to begin by assuming that we have a bounded solution involving d halfspaces. This fact will simply complicate the analysis, and so for the sake of setting the basis cases, let's assume that $W_d(1) = 1$ and $W_1(n) = n$.

For general n and d, we begin by recursively computing the optimum LP solution for the first $n-1$ halfspaces, which takes $W_d(n-1)$ time. Next, we check whether the optimum vertex v_{n-1} is feasible for the final halfspace h_n. This can be tested in just $O(d)$ time. If it is feasible, we are done. Otherwise, we need to apply the projection process and invoke a

$(d-1)$-dimensional LP on $n-1$ halfspaces. As mentioned above, it takes $O(dn)$ to perform the projection and it takes $W_{d-1}(n-1)$ time to compute the lower-dimensional LP. Up to constant factors, this yields the following recurrence:

$$W_d(n) = \begin{cases} 1 & \text{if } n = 1, \\ n & \text{if } d = 1, \\ W_d(n-1) + d + dn + W_{d-1}(n-1) & \text{otherwise.} \end{cases}$$

It is easy to show by induction that $W_d(n) = O(n^d)$. Since we don't intend to use this algorithm, here is a quick-and-dirty analysis, which will hopefully convince you that this is believable. Suppose we were to ignore the "$d + dn$" term and replace the $W_{d-1}(n-1)$ with $W_{d-1}(n)$. Then, we would have the recurrence $W_d(n) = W_d(n-1) + W_{d-1}(n)$. Does this look familiar? It is essentially the same as Pascal's famous recurrence for the binomial coefficients, $\binom{n}{d} = \binom{n-1}{d} + \binom{n}{d-1}$. It is well known that $\binom{n}{d} = O(n^d)$, and so it is not surprising that we essentially get the same asymptotic bound for our recurrence.

Considering that n may be very large, a running time of $O(n^d)$ is unacceptably slow even in $\mathbb{R}^2$. This worst-case analysis is based on the *very pessimistic* assumption that the vertex generated by the LP on the first $n-1$ constraints is *always infeasible*. It can be shown that there exist insertion order for which this happens, these are exceedingly rare. Next, we'll consider a randomized strategy.

Randomized Algorithm: Suppose that we apply the above algorithm, but we insert the halfspaces in *random order*. (As mentioned above, we need to start with d halfspaces to obtain our initial solution, but in our analysis, we will ignore this messy detail.) 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 halfspaces in $\mathbb{R}^d$, 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 $d = 1$, solve the LP by brute force in $O(n)$ time

(2) Find an initial subset of d halfspaces $\{h_1, \ldots, h_d\}$ that provide a bounded solution v_d. (If no such set exists, report that the LP is unbounded.)

(3) Randomly select a halfspace from the remaining set $\{h_{d+1}, \ldots, h_n\}$. Call this h_n. Recursively solve the LP on the remaining $n-1$ halfspaces, letting v_{n-1} denote the result. (If the LP is infeasible, then return this.)

(4) If $(v_{n-1} \in h_n)$ return v_{n-1} as the final answer

(5) Otherwise, intersect $\{h_1, \ldots, h_{n-1}\}$ with the $(d-1)$-dimensional hyperplane ℓ_n that bounds h_n and project onto $\mathbb{R}^{d-1}$. Let c' be the projection of c onto ℓ_n and then onto $\mathbb{R}^{d-1}$. Recursively solve the resulting $(d-1)$-dimensional LP with $n-1$ halfspaces. (If the LP is infeasible, then return this.) Project the optimal vertex back onto ℓ_n, and return this point.

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. In particular, make *no assumptions* about the distribution of the input. (This is an important characteristic of randomized algorithms. The performance might be affected by the random number generator, but not the input the user gives us.)

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.

Computing the Minimum (Optional): To motivate how backwards analysis works, let us consider a much simpler example, namely the problem of computing the minimum of a list of numbers. Suppose that we are given a sequence S of n distinct numbers. We permute the sequence 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, e.g., the Wikipedia entry for Harmonic Series.)

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 "backwards" 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 $T_d(n) = O(n)$, where the constant factor grows exponentially with the dimension. More precisely, we will show 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.

Recall that our algorithm was complicated by the need to start with a solution involving d halfspaces. It will simplify the analysis to ignore this technical detail. As we did in our worst-case analysis, we will start with the basis cases $T_d(1) = 1$ and $T_1(n) = n$. Our randomized analysis will depend on the random event of whether we execute the inexpensive step (4) or the expensive step (5). Given $n \geq 1$, let p_n denote the probability that the insertion of the nth hyperplane in the random order results in a change in the optimum vertex. Before we derive the value of p_n, let's see how it affects our execution time.

Case 1: With probability $(1 - p_n)$ there is no change in the optimum. It takes us $O(d)$ time to determine that this is the case (but we pay this cost irrespective of whether $v_{n-1} \in h_n$)

Case 2: With probability p_n, there is a change to the optimum. This means that we need to apply the projection process (which we saw earlier can be done in time $O(dn)$) and then invoke a $(d-1)$-dimensional LP involving $n-1$ halfspaces (which takes $T_{d-1}(n-1)$ time).

In either case, we start by solving an LP involving $n - 1$ halfspaces, which requires $T_d(n - 1)$ expected time. This suggests the following recurrence for the expected execution time:

$$T_d(n) \;=\; \begin{cases} 1 & \text{if } n = 1, \\ n & \text{if } d = 1, \\ T_d(n-1) + d + p_n(dn + T_{d-1}(n-1)) & \text{otherwise.} \end{cases}$$

The last case is the interesting one for us, since we'll be deriving an upper bound, we can simplify the last recursive term by replacing $n-1$ with n, yielding $T_d(n) \le T_d(n-1) + d + p_n(dn + T_{d-1}(n))$.

It remains is to determine what p_n is. Assuming general position, the final optimal vertex v_n is determined by the intersection of d halfspaces. This vertex is feasible with respect to all the other $n-d$ halfspaces. Since the final halfspace h_n was chosen randomly, the probability that it is among the d halfspaces that determine the final optimum is d/n. Therefore,

- With probability $p_n = d/n$, inserting h_n causes the optimum to change (see Fig. 5(b))
- With probability $1 - p_n$, inserting h_n leaves the optimum unchanged (see Fig. 5(c))

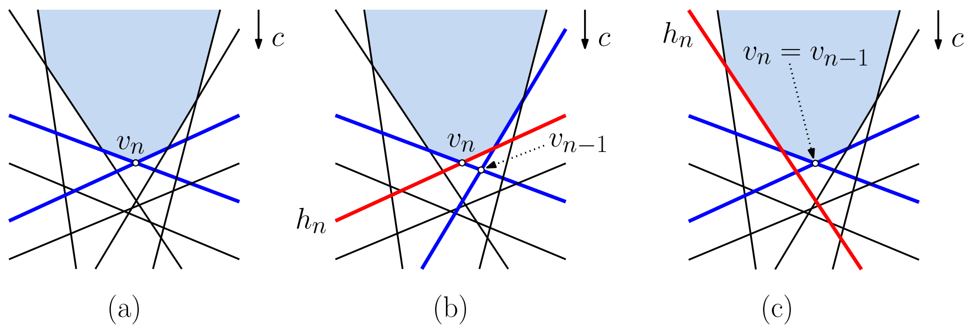

Fig. 5: Backwards analysis for the randomized LP algorithm.

Returning to our analysis, setting $p_n = d/n$ and applying our induction hypothesis (that $T_d(n) \le \gamma d! n$) we have

$$\begin{aligned} T_d(n) &\le T_d(n-1) + d + p_n\big(dn + T_{d-1}(n)\big) \\ &\le \gamma_d d!(n-1) + d + \frac{d}{n}\big(dn + (\gamma_{d-1}(d-1)!n)\big) \\ &= \gamma_d d!(n-1) + (d + d^2 + \gamma_{d-1} d!) \\ &= \gamma_d d! n + \big[d + d^2 + \gamma_{d-1} d! - \gamma_d d!\big]. \end{aligned}$$

To complete the induction proof, it suffices to show that the final term in brackets is less than zero. That is, we want to select γ_d such that

$$d + d^2 + \gamma_{d-1} d! - \gamma_d d! \le 0 \qquad \text{or equivalently} \qquad \gamma_d d! \ge d + d^2 + \gamma_{d-1} d!$$

To satisfy this we can set $\gamma_1 = 1$ and then for $d = 2, 3, \ldots$ we define

$$\gamma_d \leftarrow \frac{d + d^2}{d!} + \gamma_{d-1},$$

Recalling that d is a constant, it follows that γ_d is a constant, and therefore the final running time is $O(n)$, where the constant factor is dominated by $d!$.

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.

Why Randomization is Okay: 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? Can't we find an equally fast deterministic algorithm?

The answer to both questions is "yes". Unfortunately, the simplest deterministic algorithm is much more complex than the randomized algorithm. Also, 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!

Summary: We have presented a simple and elegant randomized incremental algorithm for solving linear programming in spaces of constant dimension. 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. In future lectures, we will see that there are numerous geometric problems that can be efficiently solved by reduction to LP (or something that is similar to LP).

Projecting Constraints: (Optional) Earlier in the lecture, we omitted discussion of the technical details on how to intersect constraint the various halfspaces h_j with the final hyperplane ℓ_n, and project the result down to $\mathbb{R}^{d-1}$ (recall Fig. 4). We mentioned there that this is essentially performing one step of Gauss elimination. Here are the technical details.

Let ℓ_n denote the hyperplane that bounds the final halfspace. We may assume it is given by the equation:

$$\ell_n : a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,d}x_d = b_n.$$

We can express this more succinctly in matrix notation. Let A_n denote the $1 \times d$ vector consisting of the i-th row of the A matrix, $A_n = (a_{n,1}, a_{n,2}, \ldots, a_{n,d})$. The inequality may be written $A_n\vec{x} = b_n$, where $\vec{x}$ is a $d \times 1$ vector and b_n is a scalar.

We want to intersect the other halfspaces with this hyperplane. Furthermore, we would like to represent the result as a LP in $d-1$ dimensional problem. (Observe that after intersection the hyperplane still resides in d space.)

The idea is to apply one step of Gauss elimination using the equation of ℓ_n to eliminate a variable from all the other inequalities. By general position, we may assume that $a_{n,1} \neq 0$. Consider an arbitrary constraint h_j that we wish to intersect with ℓ_n:

$$h_j : A_j x \leq b_j.$$

To eliminate the first dimension from h_j we multiply A_n by $(a_{j,1}/a_{n,1})$ and subtract from A_j, do the same for b_n and b_j:

$$\begin{aligned} A'_j &= A_j - \left(\frac{a_{j,1}}{a_{n,1}}\right) A_n \\ b'_j &= b_j - \left(\frac{a_{j,1}}{a_{n,1}}\right) b_n. \end{aligned}$$

To see that this works, consider an arbitrary point x on the hyperplane ℓ_n, which is equivalent to saying that $A_n x = b_n$. Suppose as well that that x satisfies constraint h_j, that is, $A_j x \leq b_j$. Then we have

$$\begin{aligned} A'_j x &= \left(A_j - \frac{a_{j,1}}{a_{n,1}} A_n\right) x \;=\; A_j x - \frac{a_{j,1}}{a_{n,1}} A_n x \\ &\leq b_j - \frac{a_{j,1}}{a_{n,1}} b_n \;=\; b'_j. \end{aligned}$$

Thus, every point that satisfies the original constraint also satisfies the modified constraint. The converse holds by a symmetrical argument. A similar elimination can be performed to the objective vector $\vec{c}$. Since the first term of each equation vanishes (is now zero), we are left with a $d-1$ dimensional problem. Reversing the process allows us to project the $d-1$ dimensional solution back into d-space.