CS261: A Second Course in Algorithms
Lecture #10: The Minimax Theorem and Algorithms for Linear Programming[*]

Tim Roughgarden[†]

February 4, 2016

1 Zero-Sum Games and the Minimax Theorem

1.1 Rock-Paper Scissors

Recall rock-paper-scissors (or roshambo). Two players simultaneously choose one of rock, paper, or scissors, with rock beating scissors, scissors beating paper, and paper beating rock.[1]

Here's an idea: what if I made you go first? That's obviously unfair — whatever you do, I can respond with the winning move.

But what if I only forced you to commit to a *probability distribution* over rock, paper, and scissors? (Then I respond, then nature flips coins on your behalf.) If you prefer, imagine that you submit your code for a (randomized) algorithm for choosing an action, then I have to choose my action, and then we run your algorithm and see what happens.

In the second case, going first no longer seems to doom you. You can protect yourself by randomizing uniformly among the three options — then, no matter what I do, I'm equally likely to win, lose, or tie. The *minimax theorem* states that, in general games of "pure competition," a player moving first can always protect herself by randomizing appropriately.

[*]©2016, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

[1]Here are some fun facts about rock-paper-scissors. There's a World Series of RPS every year, with a top prize of at least \$50K. If you watch some videos of them, you will see pure psychological welfare. Maybe this explains why some of the same players seem to end up in the later rounds of the tournament every year.

There's also a robot hand, built at the University of Tokyo, that plays rock-paper-scissors with a winning probability of 100% (check out the video). No surprise, a very high-speed camera is involved.

1.2 Zero-Sum Games

A *zero-sum game* is specified by a real-valued matrix $m \times n$ matrix $\mathbf{A}$. One player, the row player, picks a row. The other (column) player picks a column. Rows and columns are also called *strategies*. By definition, the entry a_{ij} of the matrix $\mathbf{A}$ is the row player's payoff when she chooses row i and the column player chooses column j. The column player's payoff in this case is defined as $-a_{ij}$; hence the term "zero-sum." In effect, a_{ij} is the amount that the column player pays to the row player in the outcome (i, j). (Don't forget, a_{ij} might be negative, corresponding to a payment in the opposite direction.) Thus, the row and column players prefer bigger and smaller numbers, respectively.

The following matrix describes the payoffs in the Rock-Paper-Scissors game in our current language.

	Rock	Paper	Scissors
Rock	0	-1	1
Paper	1	0	-1
Scissors	-1	1	0

1.3 The Minimax Theorem

We can write the expected payoff of the row player when payoffs are given by an $m \times n$ matrix $\mathbf{A}$, the row strategy is $\mathbf{x}$ (a distribution over rows), and the column strategy is $\mathbf{y}$ (a distribution over columns), as

$$\sum_{i=1}^{m}\sum_{j=1}^{n} \mathbf{Pr}[\text{outcome } (i,j)]\, a_{ij} = \sum_{i=1}^{m}\sum_{j=1}^{n} \underbrace{\mathbf{Pr}[\text{row } i \text{ chosen}]}_{=x_i} \cdot \underbrace{\mathbf{Pr}[\text{column } j \text{ chosen}]}_{=y_j} a_{ij}$$
$$= \mathbf{x}^\top \mathbf{A}\mathbf{y}.$$

The first term is just the definition of expectation, and the first equality holds because the row and column players randomize independently. That is, $\mathbf{x}^\top \mathbf{A}\mathbf{y}$ is just the expected payoff to the row player (and negative payoff to the second player) when the row and column strategies are $\mathbf{x}$ and $\mathbf{y}$.

In a two-player zero-sum game, would you prefer to commit to a mixed strategy before or after the other player commits to hers? Intuitively, there is only a first-mover disadvantage, since the second player can adapt to the first player's strategy. The minimax theorem is the amazing statement that *it doesn't matter.*

Theorem 1.1 (Minimax Theorem) *For every two-player zero-sum game* $\mathbf{A}$*,*

$$\max_{\mathbf{x}} \left(\min_{\mathbf{y}} \mathbf{x}^\top \mathbf{A}\mathbf{y} \right) = \min_{\mathbf{y}} \left(\max_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{y} \right). \qquad (1)$$

On the left-hand side of (1), the row player moves first and the column player second. The column player plays optimally given the strategy chosen by the row player, and the row player plays optimally anticipating the column player's response. On the right-hand side of (1), the roles of the two players are reversed. The minimax theorem asserts that, under optimal play, the expected payoff of each player is the same in the two scenarios.

For example, in Rock-Paper-Scissors, both sides of (1) are 0 (with the first player playing uniformly and the second player responding arbitrarily). When a zero-sum game is asymmetric and skewed toward one of the players, both sides of (1) will be non-zero (but still equal). The common number on both sides of (1) is called the *value* of the game.

1.4 From LP Duality to Minimax

Theorem 1.1 was originally proved by John von Neumann in the 1920s, using fixed-point-style arguments. Much later, in the 1940s, von Neumann proved it again using arguments equivalent to strong LP duality (as we'll do here). This second proof is the reason that, when a very nervous George Dantzig (more on him later) explained his new ideas about linear programming and the simplex method to von Neumann, the latter was able, off the top of his head, to immediately give an hour-plus response that outlined the theory of LP duality.

We now proceed to derive Theorem 1.1 from LP duality. The first step is to formalize the problem of computing the best strategy for the player forced to go first.

Looking at the left-hand side (say) of (1), it doesn't seem like linear programming should apply. The first issue is the nested min/max, which is not allowed in a linear program. The second issue is the quadratic (nonlinear) character of $\mathbf{x}^\top\mathbf{A}\mathbf{y}$ in the decision variables $\mathbf{x}, \mathbf{y}$. But we can work these issues out.

A simple but important observation is: the second player never needs to randomize. For example, suppose the row player goes first and chooses a distribution $\mathbf{x}$. The column player can then simply compute the expected payoff of each column (the expectation with respect to $\mathbf{x}$) and choose the best column (deterministically). If multiple columns are tied for the best, the it is also optimal to randomized arbitrarily among these; but there is no need for the player moving second to do so.

In math, we have argued that

$$\begin{aligned}\max_{\mathbf{x}}\left(\min_{\mathbf{y}} \mathbf{x}^T\mathbf{A}\mathbf{y}\right) &= \max_{\mathbf{x}}\left(\min_{j=1}^{n} \mathbf{x}^T\mathbf{A}\mathbf{e}_j\right)\\ &= \max_{\mathbf{x}}\left(\min_{j=1}^{n}\sum_{i=1}^{m} a_{ij}x_i\right),\end{aligned} \tag{2}$$

where $\mathbf{e}_j$ is the jth standard basis vector, corresponding to the column player deterministically choosing column j.

We've solved one of our problems by getting rid of $\mathbf{y}$. But there is still the nested max/min. Here we recall a trick from Lecture #7, that a minimum or maximum can often be simulated by additional variables and constraints. The same trick works here, in exactly the same way.

Specifically, we introduce a decision variable v, intended to be equal to (2), and

$$\max v$$

subject to

$$v - \sum_{i=1}^{m} a_{ij} x_i \le 0 \qquad \text{for all } j = 1, \ldots, n \tag{3}$$

$$\sum_{i=1}^{m} x_i = 1$$

$$x_1, \ldots, x_m \ge 0 \quad \text{and} \quad v \in \mathbb{R}.$$

Note that this is a linear program. Rewriting the constraints (3) in the form

$$v \le \sum_{i=1}^{m} a_{ij} x_i \qquad \text{for all } j = 1, \ldots, n$$

makes it clear that they force v to be at most $\min_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_i$.

We claim that if $(v^*, \mathbf{x}^*)$ is an optimal solution, then $v^* = \min_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_i$. This follows from the same arguments used in Lecture #7. As already noted, by feasibility, v^* cannot be larger than $\min_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_i^*$. If it were strictly less, then we can increase v^* slightly without destroying feasibility, yielding a better feasible solution (contradicting optimality).

Since the linear program explicitly maximizes v over all distributions $\mathbf{x}$, its optimal objective function value is

$$v^* = \max_{\mathbf{x}} \left(\min_{j=1}^{n} \mathbf{x}^\top \mathbf{A} \mathbf{e}_j \right) = \max_{\mathbf{x}} \left(\min_{\mathbf{y}} \mathbf{x}^\top \mathbf{A} \mathbf{y} \right). \tag{4}$$

Thus we can compute with a linear program the optimal strategy for the row player, when it moves first, and the expected payoff obtained (assuming optimal play by the column player).

Repeating the exercise for the column player gives the linear program

$$\min w$$

subject to

$$w - \sum_{j=1}^{n} a_{ij} y_j \ge 0 \qquad \text{for all } i = 1, \ldots, m$$

$$\sum_{j=1}^{n} y_j = 1$$

$$y_1, \ldots, y_n \ge 0 \quad \text{and} \quad w \in \mathbb{R}.$$

At an optimal solution $(w^*, \mathbf{y}^*)$, $\mathbf{y}^*$ is the optimal strategy for the column player (when going first, assuming optimal play by the row player) and

$$w^* = \min_{\mathbf{y}} \left(\max_{i=1}^{m} e_i^\top \mathbf{A}\mathbf{y} \right) = \min_{\mathbf{y}} \left(\max_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{y} \right). \tag{5}$$

Here's the punch line: *these two linear programs are duals.* This can be seen by looking up our recipe for taking duals (Lecture #8) and verifying that these two linear programs conform to the recipe (see Exercise Set #5). For example, the one unrestricted variable (v or w) corresponds to the one equality constraint in the other linear program ($\sum_{j=1}^{n} y_j = 1$ or $\sum_{i=1}^{m} x_i = 1$, respectively).

Strong duality implies that $v^* = w^*$; in light of (4) and (5), the minimax theorem follows directly.[2]

2 Survey of Linear Programming Algorithms

We've established that linear programs capture lots of different problems that we'd like to solve. So how do we efficiently solve a linear program?

2.1 The High-Order Bit

If you only remember one thing about linear programming, make it this:

Linear programs can be solved efficiently, in both theory and practice.

By "in theory," we mean that linear programs can be solved in polynomial time in the worst-case. By "in practice," we mean that commercial solvers routinely solve linear programs with input size in the millions. (Warning: the algorithms used in these two cases are not necessarily the same.)

2.2 The Simplex Method

2.2.1 Backstory

In 1947 George Dantzig developed both the general formalism of linear programming and also the first general algorithm for solving linear programs, the *simplex method*.[3] Amazingly, the simplex method remains the dominant paradigm today for solving linear programs.

[2]The minimax theorem is obviously interesting its own right, and it also has applications in algorithms, specifically to proving lower bounds on what randomized algorithms can do.

[3]Dantzig spent the final 40 years of his career at Stanford (1966-2005). You've probably heard the story about a student who is late to class, sees two problems written on the blackboard, assumes they're homework problems, and then goes home and solves them, not realizing that they are the major open questions in the field. (A partial inspiration for *Good Will Hunting*, among other things.) Turns out this story is not apocryphal: it was Dantzig, as a PhD student in the late 1930s, in a statistics course at UC Berkeley.

2.2.2 Geometry

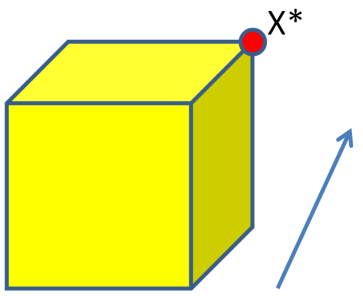

Figure 1: Illustration of a feasible set and an optimal solution x^*. We know that there always exists an optimal solution at a vertex of the feasible set, in the direction of the objective function.

In Lecture #7 we developed geometric intuition about what it means to solve a linear program, and one of our findings was that there is always an optimal solution at a vertex (i.e., "corner") of the feasible region (e.g., Figure 1).[4] This observation implies a finite (but bad) algorithm for linear programming. (This is not trivial, since there are an infinite number of feasible solutions.) The reason is that every vertex satisfies at least n constraints with equality (where n is the number of decision variables). Or contrapositively: for a feasible solution $\mathbf{x}$ that satisfies at most $n-1$ constraints with equality, there is a direction along which moving $\mathbf{x}$ continues to satisfy these constraints, and moving $\mathbf{x}$ locally in either direction on this line yields two feasible points whose midpoint is $\mathbf{x}$. But a vertex of a feasible region cannot be written as a non-trivial convex combination of other feasible points.[5] See also Exercise Set #5. The finite algorithm is then: enumerate all (finitely many) subsets of n linearly independent constraints, check if the unique point of $\mathbb{R}^n$ that satisfies all of them is a feasible solution to the linear program, and remember the best feasible solution found in this way.

The simplex algorithm also searches through the vertices of the feasible region, but does so in a smarter and more principled way. The basic idea is to use local search — if there is a "neighboring" vertex which is better, move to it, otherwise halt. The idea of neighboring vertices should be clear from Figure 1 — two endpoints of an "edge" of the feasible region. In general, we can define two different vertices to be neighboring if and only if they satisfy $n-1$ common constraints with equality. Moving from one vertex to a neighbor then just involves swapping out one of the old tight constraints for a new tight constraint; each such swap (also called a *pivot*) corresponds to a "move" along an edge of the feasible region.[6]

[4]There are a few edge cases, including unbounded or empty feasible regions, which can be handled and which we'll ignore here.

[5]Making all of this completely precise is somewhat annoying. But everything your geometric intuition suggests about these statements is indeed true.

[6]One important issue is "degeneracy," meaning a vertex that satisfies strictly more than n constraints

In an iteration of the simplex method, the current vertex may have multiple neighboring vertices with better objective function value. The choice of which of these to move to is known as a *pivot rule*.

2.2.3 Correctness

The simplex method is guaranteed to terminate at an optimal solution.[7] The intuition for this fact should be clear from Figure 1 — since the objective function is linear and the feasible region is convex, if no "local move" from a vertex is improving, then there should be no direction at all within the feasible region that leads to a better solution. Formally, the simplex method "knows that it's done" by, at termination, exhibiting a a feasible dual solution such that the complementary slackness conditions hold (see Lecture #9). Indeed, the proof that the simplex method is guaranteed to terminate with an optimal solution provides another proof of strong LP duality.

In terms of our three-step design paradigm (Lecture #9), we can think of the simplex method as maintaining primal feasibility and the complementary slackness conditions and working toward dual feasibility.[8]

2.2.4 Worst-Case Running Time

As mentioned, the simplex method is very fast in practice, and routinely solves linear programs with hundreds of thousands or even millions of variables and constraints. However, it is a bizarre mathematical fact that the worst-case running time of the simplex method is exponential in the input size. To understand the issue, first note that the number of vertices of a feasible region can be exponential in the dimension (e.g., the 2^n vertices of the n-dimensional hypercube). Much harder is constructing a linear program where the simplex method actually visits all of the vertices of the feasible region. Such an example was given by Klee and Minty in the early 1970s (25 years after simplex was invented). Their example is a "squashed" version of an n-dimensional hypercube. Such exponential lower bounds are known for all natural deterministic pivot rules.[9]

The number of iterations required by the simplex method is also related to one of the most famous open problems in combinatorial geometry, the *Hirsch conjecture*. This conjecture concerns the "diameter of polytopes," meaning the diameter of the graph derived from the

with equality. (E.g., in the plane, this would be 3 constraints whose boundaries meet at a common point.) In this case, a constraint swap can result in staying at the same vertex. There are simple ways to avoid cycling, however, which we won't discuss here.

[7]Assuming that the linear program is feasible and has a finite optimum. If not, the simplex method correctly detects which of these cases the linear program falls in.

[8]How does the simplex method find the initial primal feasible point? For some linear programs this is easy (e.g., the all-0 vector is feasible). In general, one can add an additional variable, highly penalized in the objective function, to make finding an initial feasible point trivial.

[9]Interestingly, some randomized pivot rules (e.g., among the neighboring vertices that are better, pick one at random) require, in expectation, at most $\approx 2^{\sqrt{n}}$ iterations to converge on every instance. There are now nearly matching upper and lower bounds on the required number of iterations for all the natural randomized rules.

skeleton of the polytope (with vertices and edges of the polytope inducing, um, vertices and edges of the graph). The conjecture asserts that the diameter is always at most linear (in the number of variables and constraints). The best known upper bound on the worst-case diameter of polytopes is "quasi-polynomial" (of the form $\approx n^{\log n}$), due to Kalai and Kleitman in the early 1990s. Since the trajectory of the simplex method is a walk along the edges of the feasible region, the number of iterations required (for a worst-case starting point and objective function) is at least the polytope diameter. Put differently, sufficiently good upper bounds on the number of iterations required by the simplex method (for some pivot rule) would automatically yield progress on the Hirsch conjecture.

2.2.5 Average-Case and Smoothed Running Time

The worst-case running time of the wildly practical simplex method poses a real quandary for the mathematical analysis of algorithms. Can we "correct" the theory so that it better reflects reality?

In the 1980s, a number of researchers (Borgwardt, Smale, Adler-Karp, etc.) showed that the simplex method (with a suitable pivot rule) runs in polynomial time "on average" with respect to various distributions over linear programs. Note that it is not at all obvious how to define a "random linear program." Indeed, many natural attempts lead to linear programs that are almost always infeasible.

At the start of the 21st century, Spielman and Teng proved that the simplex method has polynomial "smoothed complexity." This is like a robust version of an average-cases analysis. The model is to take a worst-case initial linear program, and then to randomly perturb it a small amount. The main result here is that, for every initial linear program, in expectation over the perturbed version of the linear program, the running time of simplex is polynomial in the input size. The take-away being that bad examples for the simplex method are both rare and isolated, in a precise sense. See the instructor's CS264 course ("Beyond Worst-Case Analysis") for much more on smoothed analysis.

2.3 The Ellipsoid Method

2.3.1 Worst-Case Running Time

The *ellipsoid method* was originally proposed (by Shor and others) in the early/mid-1970s as an algorithm for nonlinear programming. In 1979 Khachiyan proved that, for linear programs, the algorithm is actually guaranteed to run in polynomial time. This was the first-ever polynomial-time algorithm for linear programming, a big enough deal at the time to make the front page of the New York Times (if below the fold).

The ellipsoid method is very slow in practice — usually multiple orders of magnitude slower than the fastest methods. How can a polynomial-time algorithm be so much worse than the exponential-time simplex method? There are two issues. First, the degree in the polynomial bounding the ellipsoid method's running time is pretty big (like 4 or 5, depending on the implementation details). Second, the performance of the ellipsoid method on "typical cases" is generally close to its worst-case performance. This is in sharp contrast to the simplex method, which almost always solves linear programs in time far less than its worst-case (exponential) running time.

2.3.2 Separation Oracles

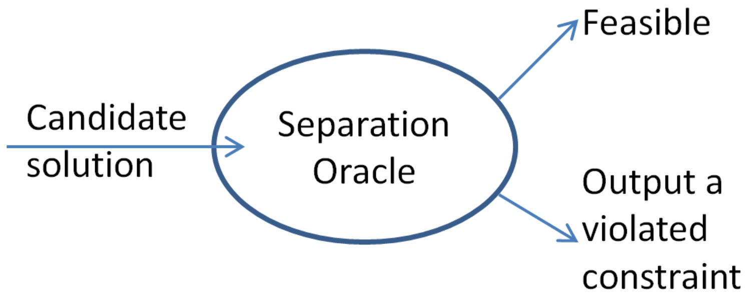

Figure 2: The responsibility of a separation oracle.

The ellipsoid method is uniquely useful for proving theorems — for establishing that other problems are worst-case polynomial-time solvable, and thus are at least efficiently solvable in principle. The reason is that the ellipsoid method can solve some linear programs with n variables and an exponential (in n) number of constraints in time polynomial in n. How is this possible? Doesn't it take exponential time just to read in all of the constraints? For other linear programming algorithms, yes. But the ellipsoid method doesn't need an explicit description of the linear program — all it needs is a helper subroutine known as a *separation oracle*. The responsibility of a separation oracle is to take as input an allegedly feasible solution $\mathbf{x}$ to a linear program, and to either verify feasibility (if $\mathbf{x}$ is indeed feasible) or produce a constraint violated by $\mathbf{x}$ (otherwise). See Figure 2. Of course, the separation oracle should also run in polynomial time.[10]

How could one possibly check an exponential number of constraints in polynomial time? You've actually already seen some examples of this. For example, recall the dual of the path-based linear programming formulation of the maximum flow problem (Lecture #8):

$$\min \sum_{e \in E} u_e \ell_e$$

[10]Such separation oracles are also useful in some practical linear programming algorithms: in "cutting plane methods," for linear programs with a large number of constraints (where the oracle is used in the same way as in the ellipsoid method); and in the simplex method for linear programs with a large number of variables (where the oracle is used to generate *variables* on the fly, a technique called "column generation").

subject to

$$\sum_{e \in P} \ell_e \geq 1 \qquad \text{for all } P \in \mathcal{P} \tag{6}$$
$$\ell_e \geq 0 \qquad \text{for all } e \in E.$$

Here $\mathcal{P}$ denotes the set of s-t flow paths of a maximum flow instance (with edge capacities u_e). Since a graph can have an exponential number of s-t paths, this linear program has a potentially exponential number of constraints.[11] But, it has a polynomial-time separation oracle. The key observation is: at least one constraint is violated if and only if

$$\min_{P \in \mathcal{P}} \sum_{e \in P} \ell_e < 1.$$

Thus, the separation oracle is just Dijkstra's algorithm! In detail: given an allegedly feasible solution $\{\ell_e\}_{e \in E}$ to the linear program, the separation oracle first checks that each ℓ_e is nonnegative (if $\ell_e < 0$, it returns the violated constraint $\ell_e \geq 0$). If the solution passes this test, then the separation oracle runs Dijkstra's algorithm to compute a shortest s-t path, using the ℓ_e's as (nonnegative) edge lengths. If the shortest path has length at least 1, then all of the constraints (6) are satisfied and the oracle reports "feasible." If the shortest path P^* has length less than 1, then it returns the violated constraint $\sum_{e \in P^*} \ell_e \geq 1$. Thus, we can solve the above linear program in polynomial time using the ellipsoid method.[12]

2.3.3 How the Ellipsoid Method Works

Here is a sketch of how the ellipsoid method works. The first step is to reduce optimization to feasibility. That is, if the objective is $\max \mathbf{c}^T\mathbf{x}$, one replaces the objective function by the constraint $\mathbf{c}^T\mathbf{x} \geq M$ for some target objective function value M. If one can solve this feasibility problem in polynomial time, then one can solve the original optimization problem using binary search on the target objective M.

There's a silly story about how to hunt a lion in the Sahara. The solution goes: encircle the Sahara with a high fence and then bifurcate it with another fence. Figure out which side has the lion in it (e.g., looking for tracks), and recurse. Eventually, the lion is trapped in such a small area that you know exactly where it is.

[11] For example, consider the graph $s = v_1, v_2, \ldots, v_n = t$, with two parallel edges directed from each v_i to v_{i+1}.

[12] Of course, we already know how to solve this particular linear program in polynomial time — just compute a minimum s-t cut (see Lecture #8). But there are harder problems where the only known proof of polynomial-time solvability goes through the ellipsoid method.

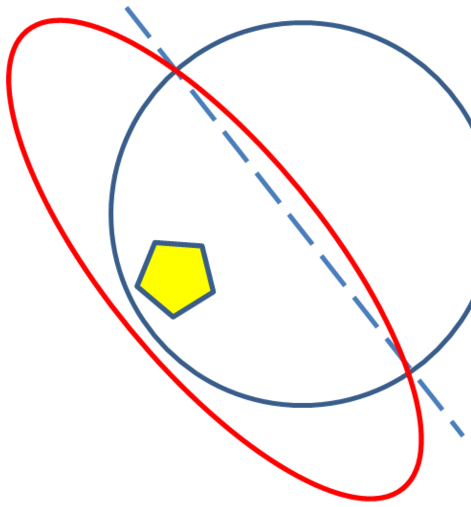

Figure 3: The ellipsoid method first initializes a huge sphere (blue circle) that encompasses the feasible region (yellow pentagon). If the ellipsoid center is not feasible, the separation oracle produces a violated constraint (dashed line) that splits the ellipsoid into two regions, one containing the feasible region and one that does not. A new ellipsoid (red oval) is drawn that contains the feasible half-ellipsoid, and the method continues recursively.

Believe it or not, this story is a pretty good cartoon of how the ellipsoid method works. The ellipsoid method maintains at all times an ellipsoid which is guaranteed to contain the entire feasible region (Figure 3). It starts with a huge sphere to ensure the invariant at initialization. It then invokes the separation oracle on the center of the current ellipsoid. If the ellipsoid center is feasible, then the problem is solved. If not, the separation oracle produces a constraint satisfied by all feasible points that is violated by the ellipsoid center. Geometrically, the feasible region and the ellipsoid center are on opposite sides of the corresponding halfspace boundary (Figure 3). Thus we know we can recurse on the appropriate half-ellipsoid. Before recursing, however, the ellipsoid method redraws a new ellipsoid that contains this half-ellipsoid (and hence the feasible region).[13] Elementary but tedious calculations show that the volume of the current ellipsoid is guaranteed to shrink at a certain rate at each iteration, and this yields a polynomial bound on the number of iterations required. The algorithm stops when the current ellipsoid is so small that it cannot possibly contain a feasible point (given the precision of the input data).

Now that we understand how the ellipsoid method works at a high level, we see why it can solve linear programs with an exponential number of constraints. It never works with an explicit description of the constraints, and just generates constraints on the fly on a "need to know" basis. Because it terminates in a polynomial number of iterations, it only ever

[13]Why the obsession with ellipsoids? Basically, they are the simplest shapes that can decently approximate all shapes of polytopes ("fat" ones, "skinny" one, etc.). In particular, every ellipsoid has a well defined and easy-to-compute center.

generates a polynomial number of constraints.[14]

2.4 Interior-Point Methods

While the simplex method works "along the boundary" of the feasible region, and the ellipsoid method works "outside in," the third and again quite different paradigm of *interior-point methods* works "inside out." There are many genres of interior-point methods, beginning with Karmarkar's algorithm in 1984 (which again made the New York Times, this time above the fold). Perhaps the most popular are "central path" methods. The idea is, instead of maximizing the given objective $\mathbf{c}^T\mathbf{x}$, to maximize

$$\mathbf{c}^T\mathbf{x} - \lambda \cdot \underbrace{f(\text{distance between } \mathbf{x} \text{ and boundary})}_{\text{barrier function}},$$

where $\lambda \geq 0$ is a parameter and f is a "barrier function" that blows up (to $+\infty$) as its argument goes to 0 (e.g., $\log \frac{1}{z}$). Initially, one sets λ so big that the problem becomes easy (when $f(x) = \log \frac{1}{z}$, the solution is the "analytic center" of the feasible region, and can be computed using e.g. Newton's method). Then one gradually decreases the parameter λ, tracking the corresponding optimal point along the way. (The "central path" is the set of optimal points as λ varies from ∞ to 0.) When $\lambda = 0$, the optimal point is an optimal solution to the linear program, as desired.

The two things you should know about interior-point methods are: (i) many such algorithms run in time polynomial in the worst case; and (ii) such methods are also competitive with the simplex method in practice. For example, one of Matlab's LP solvers uses an interior-point algorithm.

There are many linear programs where interior-point methods beat the best simplex codes (especially on larger LPs), but also vice versa. There is no good understanding of when one is likely to outperform the other. Despite the fact that it's 70 years old, the simplex method remains the most commonly used linear programming algorithm in practice.

[14]As a sanity check, recall that every vertex of a feasible region in $\mathbb{R}^n$ is the unique point satisfying some subset of n constraints with equality. Thus in principle there's always n constraints the are sufficient to describe one feasible point (given a separation oracle to verify feasibility). The magic of the ellipsoid method is that, even though a priori it has no idea which subset of constraints is the right one, it always finds a feasible point while generating only a polynomial number of constraints.