CS 224: Advanced Algorithms Spring 2017

Lecture 20 — March 30, 2017

Prof. Jelani Nelson *Scribe: Dimitris Kalimeris*

1 Overview

In the last lecture we analyzed Simplex method which was the first algorithm that was developed for solving linear programs and is still heavily used in practice.

In this lecture we will show that the correctness of the Simplex algorithm implies strong duality, and we will also prove the complementary slackness conditions. Moreover, we will comment on the running time and the theoretical vs empirical performance of the Simplex method. Then we will look at other algorithms for solving linear programs that (unlike Simplex) have provable worst-case polynomial time guarantees, namely the Ellipsoid algorithm and the interior point methods.

2 Strong Duality

We will show strong duality as a corollary of the Simplex algorithm.

Recall that we assumed that we have a linear program of the form $min\, c^T x$ subject to $Ax = b$ and $x \geq 0$. Hence, the dual program is of the form $max\, b^T y$ subject to $A^T y \leq c$ (since the primal has equality constraints the dual variables are unrestricted).

Suppose that Simplex terminates at a vertex v with basis B. Then it holds: $OPT(LP) = c^T v = c_B^T v_B$. In order to prove strong duality we would like to find a vector y that is feasible for the dual and also $c_B^T v_B = b^T y$. Remember that the Simplex algorithm implies: $v_B = A_B^{-1} b$. Thus, if we choose $y = (A_B^{-1})^T c_B$ (this y is referred to as "shadow price vector") we will have that $c_B^T v_B = b^T y$ by construction. The only thing that we need to prove is that this y is also feasible for the dual.

Define the dual slack vector $s = c - A^T y = c - A^T (A_B^{-1})^T c_B$. If $s \geq 0$ then dual feasibility obviously holds for y. Let's examine the coordinates of s that belong in B and those which don't belong separately:

- for the coordinates in B we have: $s_B = c_B - A_B^T (A_B^{-1})^T c_B = c_B - c_B = 0$
- for the coordinates not in B we have: $s_N = c_N - A_N^T (A_B^{-1})^T c_B = \widetilde{c}_N$,
  and since $\widetilde{c}_N \geq 0$ is exactly the halting condition of the Simplex algorithm $\Rightarrow s_N \geq 0$.

Hence, the shadow price vector is feasible and strong duality follows.

3 Complementary Slackness

Theorem 1. *Let x^* and y^* be the optimal solutions for the primal and the dual linear program respectively. Then $\forall j \in [n]$ it holds: $x_j = 0$, or the corresponding slack variable $s_j := \left(c - A^T y\right)_j =$*

0.

Proof. Since x^* and y^* are optimal, strong duality implies that $c^T x - b^T y = 0$. Also, because we assumed that the primal LP is in the form $min\, c^T x$, subject to $Ax = b$ and $x \geq 0$ we have that $c^T x - b^T y = \sum_{j=1}^{n} x_j s_j$. Now note that the constraints require $x_j, s_j \geq 0$, $\forall j \in [n]$. Hence since $\sum_{j=1}^{n} x_j s_j = 0 \Rightarrow x_j r_j = 0$, $\forall j \in [n]$, which completes the proof. □

4 Running Time of Simplex

The Simplex algorithm spends poly-time in every iteration, since it just performs some fixed amount of matrix/ vector operations. Thus, the running time is determined by the number of iterations. Since the algorithm will visit each vertex at most once, the number of iterations is bounded by the number of vertices which is $\binom{n}{m}$ (a vertex is determined by its basis elements). In fact, a better bound in the number of vertices is $\binom{n-\frac{m}{2}}{\frac{m}{2}}$, which follows from the McMullen's "Upper Bound Theorem" (see [1]).

It is still an open problem if there is some pivoting rule that makes Simplex run in polynomial time. However, in order for this to happen, we need to at least prove that no matter which our starting vertex is, we can still reach the optimal one in polynomially many iterations, i.e. there would never exist vertices in the polytope with the shortest path between them being super-polynomial.

Specifically, given a polytope P, defined by the constraints of an LP we can construct a graph $\mathcal{G}$ such that its set of vertices, $V(\mathcal{G})$, corresponds to the vertices of P, and the set of edges $E(\mathcal{G})$, corresponds to the neighboring vertices in P. Then, a necessary (but not sufficient) condition for Simplex to run in poly-time is that for any polytope P it holds: $diameter(\mathcal{G}(P)) \leq poly(m, n)$.

Warren Hirsch in 1957 conjectured that for any n-dimensional polytope defined as the intersection of m halfspaces it holds: $diameter(\mathcal{G}(P)) \leq m - n$. Santos disproved this conjecture more than 50 years later by showing that there exists a polytope P with $n = 43$, $m = 86$ and $diameter(P) \geq 44$ (see [2]). However, the weaker version of the Hirsch conjecture which states that $\forall P : diameter(\mathcal{G}(P)) \leq poly(m, n)$ is still open today.

Note that even if this conjecture is true it still does not imply that Simplex will run in poly-time since the algorithm can take a very long path going through exponentially many vertices before it recovers the optimal one.

5 Simplex in Practice and Smoothed Analysis

Although the worst-case running time of Simplex for any known pivoting rule is exponential, it usually runs very fast in practice. This indicates that probably the "bad instances" for Simplex are very rare, isolated and fragile.

However, worst-case analysis does not capture this intuition since we only measure the performance of an algorithm with respect to the worst possible input. Spielman and Teng in [3] introduced a variant of worst case analysis, which they named smoothed analysis. There, one takes a worst-case instance and slightly perturbs it by adding some random noise. Then, the guarantee that

is offered is that for all instances x, the $\mathbb{E}[runtime(A(\widetilde{x}))]$ is small (where we use $\widetilde{x}$ to denote the perturbed instance and the expectation is over the random noise we add), i.e. the algorithm runs fast even for instances that are very close to the worst-case. It turns out that Simplex has smoothed running time that is polynomial in n, m and the amount of noise added, which explains its empirical performance.

6 Ellipsoid Algorithm

The Ellipsoid algorithm introduced by Khachian in [4] was the first algorithm to achieve worst-case (weakly) polynomial time for linear programming. The fact that the algorithm is weakly polynomial means that its running time depends on the bit complexity (maximum number of bits) of any entry of A, b and x and not just on the number of entries.

The Ellipsoid only solves feasibility, i.e. checks whether a polytope $P = \{x \,|\, Ax \leq b\}$ is non-empty, but this is enough to solve the optimization problem as well. We can use binary search, by adding an extra constraint to the LP of the form $c^T x = K$ and decrease or increase K based on whether the new LP is feasible or not respectively.

Another way that exploits duality is to consider the polytopes $P_{primal} = \{x \,|\, Ax = b,\, x \geq 0\}$, $P_{dual} = \{y \,|\, A^T y \leq c\}$ and $P = \{(x, y) \,|\, c^T x = b^T y,\, Ax = b,\, A^T y \leq c,\, x \geq 0\}$. So if we find a feasible point in P using the Ellipsoid then, this is the optimal solution for our LP (by strong duality).

An interesting property of this algorithm is that it can solve linear programs with exponentially many or even infinite number of constraints. The only thing that this method requires in order to work, is an oracle that if you feed it with a point that is infeasible it will tell you a violated constraint (called separation oracle). Many linear programs of exponential size have separation oracles and thus can be solved efficiently.

Before we proceed with the algorithm let's define what an ellipsoid is.

Definition 2. *Given a symmetric positive definite matrix and a vector $\alpha \in \mathbb{R}^n$, we can define an ellipsoid as: $E(A, a) = \{x \in \mathbb{R}^n \,|\, (x - \alpha)^T A(x - a) \leq 1\}$.*

Note that the fact that A is a symmetric and positive definite matrix implies that there exists a matrix B, such that $A = B^T B$. Hence, we can rewrite $(x - \alpha)^T A(x - a) \leq 1$ as $||B(x - \alpha)||_2^2 \leq 1$.

The Ellipsoid algorithm is an iterative procedure:

1. start with an ellipsoid E_0 that contains P
2. at each iteration let α be the center of E_k. If $\alpha \in P$ then we are done because we found a feasible solution. Otherwise there exists at least one constraint (which defines a halfspace H) that is violated, i.e. P lies entirely on the opposite side of H from α. In that case let E_{k+1} be the minimum-volume ellipsoid containing the part of E_k that lies on the same side of H with P and repeat.
3. if at some point the volume of the ellipsoid becomes very small then we declare that the linear program is infeasible and terminate.

We state the following two lemmas without a proof that provide the worst case poly-time guarantee of the algorithm.

Lemma 3. *Given E_k and H we can compute E_{k+1} in polynomial time.*

Lemma 4. *For two consecutive ellipsoids it holds: $\frac{vol(E_{k+1})}{vol(E_k)} \leq e^{-\frac{1}{2(n+1)}}$.*

Now, if all entries of A and b have bit complexity $\leq L$ then this implies that P should be contained in a ball of appropriate radius (assuming that P is bounded). Specifically, we just need to pick E_0 to be a sphere with radius $\geq 2^{poly(n)L}$.

Define $P_{rounded} = \{(x, z) \,|\, Ax + z\vec{1} \geq b, and - 2^L \leq x_i \leq 2^L, \forall i \in [n], and - 2^L \leq z \leq 2^L\}$ (note that z is just a variable and x is a vector here). Then P is feasible if and only if there is a point $(x^*, z^*) \in P_{rounded}$ with $z^* = 0$. Note that $P_{rounded}$ is always an non-empty polytope (eg it contains the point $x = 0$, $z = 2^L$). Also, it can be proved that $volume(P_{rounded}) \geq 2^{-cL}$ for some constant c.

Hence, if we start with an E_0 for which it holds $volume(E_0) \leq c_n(radius)^n \leq 2^{poly(n)L}$, and stop when $volume(E_k) < 2^{-cL}$ we know by Lemma 4 that in each iteration the volume goes down by approximately $e^{-\frac{1}{n}}$. Thus, the number of iterations, T, till we stop satisfies: $e^{\frac{T}{n}} > 2^{poly(n)L} \Rightarrow T = poly(n)L$, which implies that the Ellipsoid algorithm is weakly polynomial.

7 Interior Point Methods

The interior point methods were introduced in 1984 by Karmarkar [5] and they follow the opposite approach from Simplex. Instead of starting at a vertex and walk on neighboring vertices till we find the optimal one, we start deep in the polytope P, and we gradually move towards the optimal vertex (always staying at the interior of P).

Assume that we have a linear program of the form $min\, c^T x$ subject to $Ax \geq b$ and $x \geq 0$. Then we can define the slackness vector to be: $s(x) := Ax - b$. Our goal here is to compute $min_{x \in \mathbb{R}^n} f_\lambda(x)$, where $f_\lambda(x) = \lambda \cdot c^T x + p(s(x))$. For the function p (called the barrier function) it holds: $p \to \infty$ if $\exists i : \big(s(x)\big)_i \to 0$. The idea is that at the beginning λ will be very small and hence in order to minimize f_λ we need to be as slack as possible. However, the value of λ will start to increase in the next iterations and thus the p term will become much less significant and we will end up focusing on minimizing $c^T x$.

The (very high-level) iterative procedure is as follows:

1. find x in the interior of $P_{rounded}$ (as defined in the previous section)
2. obtain an approximate minimizer $\widetilde{x}(\lambda_0)$ for f_{λ_0} for some very small λ_0
3. for $k = 1$ to k^* run a few iterations of Newton's method in order to turn $\widetilde{x}(\lambda_k)$ into a "great" solution for λ_{k+1}, yielding $\widetilde{x}(\lambda_{k+1})$
4. output $\widetilde{x}(\lambda_{k^*})$

In the next lecture we will analyze the interior point methods in more detail.

References

[1] Peter McMullen. The maximum numbers of faces of a a convex polytope. Mathematika 17:179-184, 1971.

[2] Francisco Santos. A counterexample to the Hirsch conjecture, Annals of Mathematics 176 (10: 383-412, doi:10.4007/annals.2012.176.1.7, 2011.

[3] Daniel A. Spielman, Shang-Hua Teng. Smoothed analysis of algorithms. Why the simplex algorithm usually takes polynomial time. J. ACM 51(3): 385-463, 2004.

[4] Leonid Khachian. A polynomial algorithm in linear programming. Doklady Akademi Nauk USSR 244:1093-1096, 1979.

[5] Narendra Karmarkar. A New Polynomial Time Algorithm for Linear Programming. Combinatorica, Vol 4, 4:373-395, 1984.