15-854: Approximations Algorithms **Lecturer:** R.Ravi
Topic: Linear Programming **Date:** 10/26/2005
Scribe: David Abraham

14.1 Introduction

Suppose we are trying to prove that algorithm ALG is an α-approximation problem for some minimization problem [1]. In Lecture 1, we were given the following template:

1. Find a lower bound LB on the cost of OPT - i.e. $LB(I) \leq c(OPT(I))$ for all instances I.
2. Prove $c(ALG(I)) \leq \alpha LB(I)$, for all instances I.
3. Conclude $c(ALG) \leq \alpha LB \leq \alpha c(OPT)$

The second step requires us to prove a relationship between ALG and LB. Sometimes, we might think of an algorithm for which we know of no related lower bounds. Other times, we might think of a lower bound, but no related algorithm.

In this lecture, we show how *linear programming* can be used as a general technique to help us (i) find lower bounds for minimization problems, and (ii) derive algorithms related to these lower bounds.

14.2 Linear Programming as a Lower Bounding Technique

In this section, we will show how to use linear programming to find lower bounds for minimization problems. We will use the Weighted Vertex Cover (WVC) problem as an extended example.

An instance of the WVC problem consists of a graph $G = (V, E)$, together with a non-negative cost function $c : V \to \Re^+$ on its vertices. A subset C of V is a called a *cover* if all edges in E have at least one endpoint in C. The *cost* of a cover C is the total cost of its vertices. The WVC problem is to find a cover with minimum cost.

14.2.1 Step 1: Reduce Problem to an Integer Program

Given an instance of Weighted Vertex Cover, we can encode it as an *integer programming* (IP) problem. Create a 0/1 indicator variable x_v for each vertex $v \in V$: $x_v = 1$ indicates v is in the cover; $x_v = 0$ otherwise. We force $\{v : x_v = 1\}$ to be a cover by requiring that for each edge $\{u, v\} \in E$, $x_u = 1$ or $x_v = 1$ (or equivalently, $x_u + x_v \geq 1$). Finally, since our objective is to find a *minimum*-cost vertex cover, we get the following IP:

[1] For exposition purposes, these notes only deal with minimization problems. All techniques discussed are analogously applicable to maximization problems

$$\begin{aligned} \text{OBJECTIVE FUNCTION:} \quad & \min \sum_{v \in V} c(v) x_v \\ \text{CONSTRAINTS:} \quad & x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \\ & x_v \in \{0, 1\} \qquad \forall v \in V \end{aligned}$$

More generally, an integer program consists of:

- a set $x = \{x_1, x_2, \ldots, x_n\}$ of n variables, each of which can only take on a *integral* values.
- m *linear* inequalities over x.
- an optional *objective function* to minimize some linear function of x.

A *feasible solution* to an IP is an assignment of values to variables that satisfies (i) the integrality constraints (variables can only take on integral values), and (ii) the m linear constraints. The integer programming feasibility problem is to find a feasible solution, or determine that no such solution exists.

An *optimal solution* Z^*_{IP} to an IP is a feasible solution that minimizes the objective function. The integer programming optimization problem is to find an optimal solution, or determine that no feasible solution exists.

Note two important properties of the program above: (i) any feasible solution corresponds to a cover, and (ii) the cost of the corresponding cover is just the value of the objective function. Many NP-hard optimization problems can be encoded as an IP optimization problem such that these properties hold - i.e. a feasible solution to the IP corresponds to a feasible solution to Π, and the cost of the solution for Π is the value of the objective function. Finally, we remark that IP optimization is NP-hard, as can be seen by reducing from the decision version of WVC.

14.2.2 Step 2: Relax Integer Program to Linear Program and obtain Lower Bound

The integer program above constrains each variable x_v to the value 0 or 1. We can *relax* these constraints by only requiring that (i) $x_v \geq 0$, and (ii) $x_v \leq 1$. All constraints are now linear functions over x, and so this is a *linear program* (LP).

$$\begin{aligned} \text{OBJECTIVE FUNCTION:} \quad & \min \sum_{v \in V} c(v) x_v \\ \text{CONSTRAINTS:} \quad & x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \\ & 0 \leq x_v \leq 1 \qquad \forall v \in V \end{aligned}$$

Any feasible solution to an integer program is also a feasible solution to its corresponding relaxed linear program. Hence, $c(Z^*_{LP}) \leq c(Z^*_{IP})$, where Z^*_{LP} is an optimal solution to the relaxed linear program. So Z^*_{LP} gives a lower bound on $OPT = Z^*_{IP}$.

14.3 Linear Programming as an Approximation Algorithm

Linear programming is polynomial-time solvable (see Section 14.4.3). In this section, we will see one way of using linear programming as the basis of an approximation algorithm. The main difficulty is that an LP may return a *fractional* solution, when our original IP encoding only makes sense for integral solutions. Hence, we will try to *round* the fractional solution to an integral solution, and hope that, in the process, we don't increase the value of the objective function by too much. Here is one rounding algorithm for WVC.

1. Construct the integer programming encoding of WVC.
2. Relax the IP to get a linear program.
3. Solve the LP to get an optimal solution Z^*_{LP}.
4. For each $x_v \geq \frac{1}{2}$, round x_v up to 1.
5. For each $x_v < \frac{1}{2}$, set round x_v down to 0.
6. Output $\{v : x_v = 1\}$.

We remark that that Z^*_{LP} must exist, since the vertex set V of the input graph is itself a cover, and so there is at least one feasible solution to the LP. Hence, this algorithm always returns a vertex set, which we denote by ALG. We will show that ALG forms a vertex cover and that $c(ALG) \leq 2c(OPT)$. Since the algorithm runs in polynomial time, this makes it a 2-approximation for the WVC problem.

Since Z^*_{LP} is a feasible solution to the LP, we have for each edge $\{u, v\} \in E$, $x_u + x_v \geq 1$. It follows that $x_u \geq \frac{1}{2}$ or $x_v \geq \frac{1}{2}$. Hence, for every edge $\{u, v\}$, the algorithm sets at least one of x_u and x_v to 1, and so this too is a feasible solution to the LP. In fact, since this assignment is integral, it is also a feasible solution to the IP. Therefore, ALG is a vertex cover.

We now show that this rounding process does not increase the value of the objective function by too much.

$$
\begin{aligned}
c(ALG) &= \sum_v c(v) x_v(ALG) \\
&= \sum_{v: x_v(ALG)=1} c(v) \\
&= \sum_{v: x_v(Z^*_{LP}) \geq \frac{1}{2}} c(v) \\
&\leq 2 \sum_{v: x_v(Z^*_{LP}) \geq \frac{1}{2}} c(v) x_v(Z^*_{LP}) \\
&\leq 2c(Z^*_{LP})
\end{aligned}
$$

From the work above, we know $c(Z^*_{LP}) \leq c(OPT)$. Hence, by step 3 of the template from Lecture 1, we have $c(ALG) \leq 2c(Z^*_{LP}) \leq 2c(OPT)$. Therefore, this algorithm is a 2-approximation for the WVC problem.

14.3.1 Integrality Gap

A natural question to ask is what is the best approximation guarantee we can hope for using Z^*_{LP} as a lower bound? The question is answered by the *integrality gap*, which is defined for problem Π as:

$$
\sup_{instances\ I\ of\ \Pi} \left(\frac{Z^*_{IP}(I)}{Z^*_{LP}(I)} \right)
$$

We will prove a lower bound on the integrality gap of WVC. Consider the graph K_n with $c(v) = 1$ for each vertex v. Any vertex cover of K_n must include at least $n-1$ vertices, and in fact any subset of $n-1$ vertices is an optimal cover with cost $n-1$. However, one feasible solution to the relaxed LP formulation is to just set each $x_v = \frac{1}{2}$ giving a total cost of $\frac{n}{2}$. The integrality gap is therefore at least $\frac{n-1}{\frac{n}{2}} = 2(1 - \frac{1}{n})$ which tends to 2 as n tends to infinity.

The LP-based algorithm above has an approximation ratio of 2. Hence, it makes the best asymptotic use possible of the LP lower bound.

14.4 Algorithms for Solving Linear Programming

In this section, we will briefly discuss three algorithms for solving linear programming. We are interested in the worst-case running time of these algorithms, since we would like to use them as subroutines in our approximation algorithms. So before moving on to this discussion, we firstly need to define the size of an LP.

14.4.1 Linear Program Size

Consider the following LP with n variables $x_1, x_2, \ldots, x_n$ and m constraints.

$$\begin{aligned} \text{OBJECTIVE FUNCTION:} \quad & \min c_1x_1 + c_2x_2 + \ldots c_nx_n \\ \text{CONSTRAINTS:} \quad & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \geq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \geq b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \geq b_m \end{aligned}$$

The size of this program is clearly $O(poly(n, m, \lg(\max a_{ij}, b_i, c_j))$. We will see in Section 14.4.3 that linear programming is in polynomial time with respect to this size [2]. Note that if we encode a problem Π as a linear program, the size of the linear program may not be polynomial in the size of Π. So even though we can solve the linear program in polynomial-time with respect to its size, this may not be polynomial time with respect to the size of Π. In Section 14.4.3, we will see that we can sometimes efficiently solve LPs that are exponentially large in the size of Π.

14.4.2 Simplex Algorithm

Dantzig's simplex algorithm was the first practical algorithm for linear programming. The algorithm makes use of the following nice geometric interpretation of an LP.

We can think of each constraint as a half-space in n-dimensions, where each dimension corresponds to a variable. The intersection of the m half-spaces contains exactly the set of feasible solutions. Note that the intersection is *convex* - if two points are in the intersection, then so too is their midpoint. We call this intersection a convex *polytope*. The LP feasibility problem is equivalent to finding a point in the polytope. The LP optimization problem is to feasible solution minimizing the objective function. Since the intersection is convex, it must be the case that an optimal solution occurs at a vertex of the polytope. Furthermore, a vertex is an optimal solution if and only if no adjacent vertex is a better solution.

The basic simplex algorithm starts with a vertex of the polytope. While there is some neighbor of this vertex with lower objective value, the algorithm moves to this vertex and repeats.

The *pivot rule* selects the next vertex to which we move. There are several natural candidate rules (for example, random or lowest objective value). The choice of rule can significantly impact the running time of the algorithm.

Although the simplex algorithm is fast in practice, it is known to take exponential time in the worst case (an n-dimensional polytope with m faces has $O(\binom{m}{n})$ vertices). Researchers have concentrated on characterizing LPs that are efficiently solvable by the simplex algorithm, as well as modifying the simplex algorithm in an effort to make it run in worst-case polynomial time.

[2] It is not known if linear programming is strongly polynomial-time solvable (i.e. solvable in $O(poly(n, m))$).

14.4.3 Ellipsoid Algorithm

The Ellipsoid algorithm, invented by Khachiyan [2], is the first known polynomial-time algorithm for linear programming. The basic algorithm only solves the feasibility problem. However, by including an upper and lower bounds on the objective function as constraints, we can solve the optimization problem using binary search.

To solve the feasibility problem, we begin by constructing an ellipsoid E that completely encloses the feasible polytope. Next, we compute the center C of E and test if it violates any of the m constraints. If not, we have found a feasible solution. Otherwise, one of the constraints if violated, and so we know a hyperplane H that lies between C and the polytope. Now, find the hyperplane H' parallel to H which intersects C. This defines a half-ellipsoid containing the polytope. It is possible to efficiently find a new ellipsoid E' that contains this half-ellipsoid, with size at most $(1 - \frac{1}{n})$ of the original ellipsoid E. We can now repeat on E', terminating whenever we find a feasible solution. After a suitably large polynomial number of iterations, if no solution has been found, it is possible to terminate and claim that there is no feasible solution. For more details, see [2].

14.4.3.1 Exponential Number of Constraints and Separation

A linear program is *compact* if its size is polynomial in the size of the problem it is encoding. For some problems, we may not know be able to find a compact linear programming encoding.

For example, given a weighted directed graph $G = (V, A)$, consider the problem of finding a minimum-cost directed spanning tree. We can encode the decision to include an edge e in the solution as a 0/1 variable x_e. Because we seek a tree, it seems natural to have a *cycle-elimination* requirement: for each cycle $C = (e_1, e_2, \ldots, e_k, e_1)$ of G, $x_{e_1} + x_{e_2} + \ldots + x_{e_k} \leq k - 1$. The problem with this is that a directed graph may admit an exponential number of cycles, and hence the LP relaxation for this problem has an exponential number of constraints.

Recall that each iteration of the Ellipsoid algorithm either determines that the center is feasible, or returns a constraint that it violates. We call this task the *separation* problem. One way to solve the separation problem is to check the center against each constraint individually. In the spanning tree problem above, each iteration could therefore take exponential time. Consider the following alternative separation algorithm:

Given a fractional assignment of values to variables x_e, construct the graph $G = (V, A)$ and assign each edge e the length $l(e) = 1 - x_e$. Now use dynamic programming to compute a table d, where $d[u][v][k]$ is the shortest length path in G from u to v using exactly k edges. (This program is a trivial adaptation of Bellman-Ford's algorithm.) We claim that the assignment is a feasible solution if and only if $d[u][u][k] \geq 1$ for all $u \in V$ and $k > 0$.

Let $C = (e_1, e_2, \ldots, e_k, e_1)$ be an arbitrary k-length cycle of G. If the assignment is feasible, then $x_{e_1} + x_{e_2} + \ldots + x_{e_k} \leq k - 1$. Hence, $1 < k - (x_{e_1} + x_{e_2} + \ldots + x_{e_k}) = (1 - x_{e_1}) + (1 - x_{e_2}) + \ldots + (1 - x_{e_k})$, and so the length of this cycle in G is at least 1. Alternatively, if $x_{e_1} + x_{e_2} + \ldots + x_{e_k} > k - 1$, then $1 > (1 - x_{e_1}) + (1 - x_{e_2}) + \ldots + (1 - x_{e_k})$, and so the length of this cycle is strictly less than 1.

Hence, we can solve the separation problem by running this polynomial-time dynamic program. If there is no $d[u][u][k] < 1$ for $k > 0$, then all cycle-elimination constraints are satisfied. Otherwise,

$d[u][u][k] < 1$ describes a cycle-elimination constraint that is violated.

In summary, if we can solve the separation problem in polynomial time, we can use the ellipsoid algorithm to efficiently solve LP optimization problems with an exponential number of constraints.

14.4.4 Interior-Point Algorithm

Although the ellipsoid algorithm runs in polynomial time, it is not efficient in practice. The first polynomial-time algorithm discovered for linear programming that is also fast in practice was given by Karmarkar [1]. Like the simplex algorithm, Karmarkar's interior-point algorithm maintains a feasible solution, which is successively updated on each iteration. However, instead of moving from vertex to vertex, the algorithm traverses through the interior of the polytope.

14.5 Conclusion

In this lecture, we covered the following technique for approximating minimization problems: Convert the problem into an integer program. Relax the integer program to obtain a linear program. Solve the linear program to obtain a lower bound on the optimal solution to the original problem. Round the linear program to obtain an integral solution, and prove that the this is not too far from optimal.

We also briefly covered three different algorithms for solving linear programming. The ellipsoid algorithm runs in time polynomial in the size of the LP. If the encoding of our minimization problem Π is exponential in size, we saw that the ellipsoid algorithm can still run in time polynomial in the size Π as long as we can provide a polynomial-time separator.

References

[1] N. Karmarkar. *A New Polynomial-Time Algorithm for Linear Programming.* Combinatorica 4, 373-395, 1984.

[2] L. Khachiyan. *A Polynomial algorithm in Linear Programming.* Doklady Akademii Nauk SSSR 244, 1093 - 1096, 1979.