CS4234: Optimisation Algorithms Lecture 2

Introduction to LP

V1.0: Seth Gilbert, V1.1: Steven Halim *August 10, 2016*

Abstract

Today we introduce Linear Program(ming) (LP), a powerful tool for solving optimization problems. We discuss some terminology and notation, and we give some examples of how to use LPs to solve basic optimization problems. Then we talk about Integer Linear Programs (ILPs), and discuss how to relax ILPs to solve Combinatorial Optimization Problems that require integer answers. We use MIN-WEIGHT-VERTEX-COVER as an example, giving a 2-approximation algorithm.

1 Linear Programming

Linear Program (or Linear Programming, both usually abbreviated as LP) is a general and very powerful technique[1] for solving optimization problems where the objective (i.e., the thing being optimized) and the constraints are *linear*. Out in the real world, this is the standard approach for solving the Combinatorial Optimization Problems (COPs) that arise all the time. This technique is so common that an LP solver is now included in most common spreadsheets, e.g., Excel. (Note that the term "programming" refers not to a computer program, more to a program in the sense of an "event program," i.e., a plan for something.)

A typical Linear Program consists of three components:

- A list of (real-valued) variables $x_1, x_2, \ldots, x_n$. The goal of your optimization problem is to find good values for these variables.
- An objective function $f(x_1, x_2, \ldots, x_n)$ that you are trying to maximize or minimize. The goal is to find the best values for the variables so as optimize this function.
- A set of constraints that limits the feasible solution space. Each of these constraints is specified as an inequality.

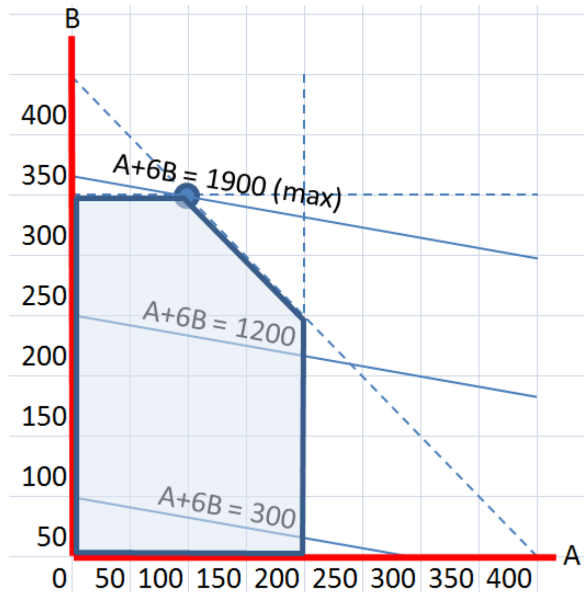
In a Linear Programming problem, both the objective function and the constraints are *linear* functions of the variables.

Example 1. You are employed by Acme Corporation, a company that makes two products: widgets and bobbles. Widgets sell for 1 SGD/widget, and bobbles sell for 6 SGD/bobble. Bobbles clearly make more money for you, and so ideally you would like to sell as many bobbles as you can. However, after doing some market research, you have discovered that there is only demand for at most 300 bobbles and at most 200 widgets. It also turns out that your factory can only produce at most 400 units, whether they are widgets or bobbles. How many widgets and bobbles should you make, in order to maximize your total revenue?

We will represent this as a Linear Programming problem. This problem has two variables: A and B. The variable A represents the number of widgets and the variable B represents the number of bobbles. Your revenue is equal to $A + 6B$, i.e., 1/widget and 6/bobble. Your goal is to maximize this revenue. Your constraints are that $A \leq 200$ and $B \leq 300$: that represents the demand constraints from the market. Your other constraint is a supply constraint: $A + B \leq 400$, since you can only make 400 units total. Finally, we will include two more constraints that are obvious: $A \geq 0$ and $B \geq 0$. These were not included in the problem, but are important to exclude negative solutions. Put together, this yields the following Linear Program:

[1] It is taught in Business School too!

$$
\begin{array}{rcll}
\max{(A+6B)} & & & \text{where:} \\
A & \leq & 200 & \\
B & \leq & 300 & \\
A+B & \leq & 400 & \\
A & \geq & 0 & \\
B & \geq & 0 &
\end{array}
$$



On the left, is the LP represented mathematically, specified in terms of an objective function and a set of constraints. On the right is a picture representing the LP geometrically, where the variable A is drawn as the x-axis and the variable B is drawn as the y-axis.

The dashed lines here represent the constraints: $A \leq 200$ (i.e., a vertical line), $B \leq 300$ (i.e., a horizontal line), and $A + B \leq 400$ (i.e., the diagonal line). Each constraint defines a ***halfspace***, i.e., it divides the universe of possible solutions in half. In two-dimensions, each constraint is a line. In higher dimensions, a constraint is defined by a hyperplane[2].

Everything that is beneath the three lines represents the ***feasible region***[3], which is defined as the values of A and B that satisfy all the constraints. In general, the feasible region is the intersection of the halfspaces defined by the hyperplanes, and from this we conclude that the feasible region is a convex[4] polygon.

Definition 1 *The **feasible region** for a Linear Program with variables $x_1, x_2, \ldots, x_n$ is the set of points $(x_1, x_2, \ldots, x_n)$ that satisfy all the constraints.*

Notice that the feasible region for a Linear Program may be: (i) empty, (ii) a single point, or (iii) infinite.

For every point in the feasible region, we can calculate the value of the objective function: $A + 6B$. The goal is to find a point in the feasible region that maximizes this objective. For each value of c, we can draw the line for $A + 6B = c$. Our goal is to find the maximum value of c for which this line intersects the feasible region. You can see, in the picture above, we have drawn in this line for three values of c: $c = 300$, $c = 1\,200$, and $c = 1\,900$. The last line, where $A + 6B = 1\,900$ intersects the feasible region at exactly one point: $(100, 300)$. This point, then, is the maximum value that can be achieved.

One obvious difficulty in solving LPs is that the feasible space may be infinite, and in fact, there may be an infinite number of optimal solutions. (Remember, we are considering real numbers here, so any line segment has a infinite number of points on it.) Let's think a little bit about whether we can reduce this space of possible solutions.

Imagine that I give you possible solution $(100, 300)$ and ask you to decide if it maximizes the objective function. How might you decide?

[2]It will be (much) harder to visualize an LP on more than 3 dimensions. Most textbook examples are on 2D (2 variables).
[3]In [1], this feasible region is called a 'simplex', hence the name of Simplex Method in Section 2.1.
[4]Mathematically, for each pair of points in the set, the line joining the points is wholly contained in the set.

1. You draw the geometric picture and look at it. Recall that $f(100, 300) = 1\,900$. The line represented by the objective function $A + 6B = 1\,900$ cannot move up any farther. Thus, clearly 1 900 is the maximum that can be achieved and hence $(100, 300)$ is a maximum.

2. Maybe we can prove algebraically that $(100, 300)$ is maximal. Recall, one of the constraints shows that $A+B \leq 400$. We also know that $B \leq 300$, and so $5B \leq 1\,500$. Putting these facts together, we conclude that:

$$\begin{array}{rcl} A + B & \leq & 400 \\ 5B & \leq & 1\,500 \\ \hline A + 6B & \leq & 1\,900 \end{array}$$

   Since the objective function is equal to $A + 6B$, this shows that we cannot possibly find a solution better than 1 900. Since we have found such a solution, we know that $(100, 300)$ is optimal.

   This may seem like a special case, but in fact it turns out that you can always generate such a set of equations to prove that you have solved your LP optimally!

3. One important fact about Linear Programs is that this maximum is always achieved at a vertex of the polygon defined by the constraints (if the feasible region is not empty). Notice that there may be other points (e.g., on an edge or a face) that also maximize the objective, but there is always a vertex that is at least as good. (We will not prove this fact today, but if you think about the geometric picture, it should make sense.) Therefore, one way to prove that your solution is optimal is to examine all the vertices of the polygon.

   How many vertices can there be? In two dimensions, a vertex may occur wherever two (independent) constraints intersect. In general, if there are n dimensions (i.e., there are n variables), a vertex may occur wherever n (linearly independent) hyperplanes (i.e., constraints) intersect. Recall that if you have n linearly independent equations and n variables, there is a single solution—that solution defines a vertex. Of course, if the equations are not linearly independent, you may get many solutions—in that case, there is no vertex. (For example, if the constraints define hyperplanes that are parallel and do not intersect, there is no vertex.) Or, alternatively, if the intersection point is outside the feasible region, this too is not a vertex.

   So in a system with m constraints and n variables, there are $\binom{m}{n} = O(m^n)$ vertices.

   We have thus discovered an exponential time $O(m^n)$ time algorithm for solving a Linear Program: Enumerate each of the $O(m^n)$ vertices of the polytope, calculate the value of the objective function for each point, and take the maximum.

2 Solving Linear Programs

In this class, for the most part, we will ignore the problem of solving Linear Programs. There exist fast and efficient LP solvers, and we will rely on these as a *black-box.* Please study Chapter 29 of [1] if you are keen to explore more details. However, here we give a few hints as to how these LP solvers work. (It is a fascinating and well-studied problem, and if you can build a faster LP solver, you can make a lot of money!)

2.1 Simplex Method

One of the earliest techniques for solving an LP—and still one of the fastest today—is the Simplex method. It was invented by Dantzig in 1947, and remains in common use today. There are many variants, but all take exponential time in the worst-case. However, in practice, for almost every LP that anyone has ever generated, it is remarkably fast.

The basic idea behind the Simplex method is remarkably simple. Recall that if an LP is feasible, its optimum is found at a vertex. (Again, remember that the optimum may be achieved at other points as well, but this does not matter to us.) Hence, the basic algorithm can be described as follows, where the function f represents the objective function.

1. Find any (feasible) vertex v.
2. Examine all the neighboring vertices of v: $v_1, v_2, \ldots, v_k$.
3. Calculate $f(v)$, $f(v_1), f(v_2), \ldots, f(v_k)$. If $f(v)$ is the maximum (among its neighbors), then stop and return v.
4. Otherwise, choose *one of* the neighboring vertices v_j where $f(v_j) > f(v)$. Let $v = v_j$.
5. Go to step (2).

There are several things to notice about this algorithm. First, if the algorithm stops in step (3), then it really has found an optimum point. To prove this fact, we need to examine the geometry of the polytope; because the feasible region is convex and the objective function linear, we can conclude that if $f(v)$ is maximum among its neighbors, then it is maximum in the entire feasible region. This fact ensures that the Simplex Method always returns the right answer.

Second, observe that it will eventually terminate. At some point, it will have explored all the vertices of the polytope. Recall that at one of the vertices we will find an optimum, and hence we will eventually terminate. This also bounds the worst-case running time as $O(m^n)$ for an LP with n variables and m constraints.

Third, notice that Step (4) is somewhat underspecified: Which neighboring vertex should we choose? Depending on how we choose the neighboring vertex, we can get very different performance. (In the case where $n = 2$, there are not many choices. But as the dimension grows, there become a much larger number of neighboring vertices to choose among.) The rule for choosing the next vertex is known as the *pivot rule*, and a large part of designing an efficient simplex implementation is choosing the pivot rule. Even so, all known pivot rules take worst-case exponential time.

Fourth, implementing this algorithm efficiently requires some care. How should one find neighboring vertices and determine where to go next (without re-calculating all the vertices in every step)? In fact, using some basic Linear Algebra and matrix manipulation[5], this can be implemented quite efficiently (e.g., using techniques like Gaussian elimination).

2.2 An Example

As an example, consider running the Simplex Method on Example 1 above (i.e., the Acme corporation optimization problem). In this case, the Simplex method might start with the feasible vertex $(0, 0)$.

First iteration. In the first iteration, it would calculate $f(0, 0) = 0$. It would also look at the two neighboring vertices, calculating that $f(0, 300) = 1\,800$ and $f(200, 0) = 200$. Having discovered that $(0, 0)$ is not optimal, it would choose one of the two neighbors. *Assume*, in this case, that the algorithm chooses next to visit neighbor $(200, 0)$[6].

Second iteration. In the second iteration, it would calculate[7] $f(200, 0) = 200$. It would also look at the two neighboring vertices, calculating that $f(0, 0) = 0$ and $f(200, 200) = 1\,400$. In this case, there is only one neighboring vertex that is better, and it would move to $(200, 200)$.

[5] Hence we need MA1101R as pre-requisite for this module.
[6] Notice that it does not greedily choose the best local move at all times.
[7] Notice that we have computed this value before. Some form of memoization can be used to avoid re-computation.

Third iteration. In the third iteration, it would calculate $f(200, 200) = 1\,400$. It would also look at the two neighboring vertices, calculating that $f(200, 0) = 200$ and $f(100, 300) = 1\,900$. In this case, there is only one neighboring vertex that is better, and it would move to $(100, 300)$.

Fourth iteration. In the fourth iteration, it would calculate $f(100, 300) = 1\,900$. It would also look at the two neighboring vertices, calculating that $f(200, 200) = 1\,400$ and $f(0, 300) = 1\,800$. After discovering that $(100, 300)$ is better than any of its neighbors, the algorithm would stop and return $(100, 300)$ as the optimal point.

Notice that along the way, the algorithm might calculate some points that were not vertices. For example, in the second iteration, it might find the point $(400, 0)$—which is not feasible. Clearly, a critical part of any good implementation is quickly calculating the feasible neighboring vertices.

For the remainder of today, we will ignore the problem of *how to solve* Linear Programs, and instead focus on *how to use* Linear Programming to solve combinatorial graph problems. For now, here's what you need to know about solving Linear Programs:

- If you can represent your Linear Program in terms of a polynomial number of variables and a polynomial number of constraints, then there exist polynomial time algorithms for solving them.
- You can find an LP solver in Excel to experiment with.
- The existing LP-solvers are very efficient for almost all the LPs that you would want to solve. You can try Stanford's ACM ICPC Simplex code library[8].

3 MIN-WEIGHT-VERTEX-COVER and Integer Linear Program

We now introduce the *weighted* version of the MIN-VERTEX-COVER problem, and show how to represent it as an *Integer* Linear Program (ILP). We can then *relax* the ILP, leading to a Linear Program that we can solve. Finally, we use the LP to construct a 2-approximation algorithm.

3.1 MIN-WEIGHT-VERTEX-COVER

To this point, we have looked at *unweighted* version of MIN-VERTEX-COVER: Every vertex has the same (unit) weight 1. It is natural to consider a weighted version of the problem where different vertices have different weights.

Definition 2 *A* **MIN-WEIGHT-VERTEX-COVER** *for a graph $G = (V, E)$ where each vertex $v \in V$ has* ***weight*** *$w(v)$, is a set $S \subseteq V$ such that for every edge $e = (u, v) \in E$, either $u \in S$ or $v \in S$. The cost of vertex cover S is the sum of the weights, i.e., $\sum_{v \in S} w(v)$.*

For MIN-WEIGHT-VERTEX-COVER problem, the 2-approximation algorithm (both the randomized and the deterministic variants) presented in the previous lecture is no longer sufficient. See the examples in Figure 1. In both examples, the simple deterministic 2-approximate algorithm examines exactly one edge and adds both endpoints, incurring a cost of 101. And yet in both cases, there are much better solutions, of cost 9 in the first case and of cost 1 in the second case. Therefore, the deterministic 2-approximate algorithm for MIN-VERTEX-COVER is *no longer* a 2-approximation algorithm for the weighted variant.

There are several methods for solving the MIN-WEIGHT-VERTEX-COVER problem, and our goal over the rest of this lecture is to develop one such solution. In order to do so, we will use Linear Programming that we have been exposed with earlier.

[8]See `https://github.com/jaehyunp/stanfordacm/blob/master/code/Simplex.cc`

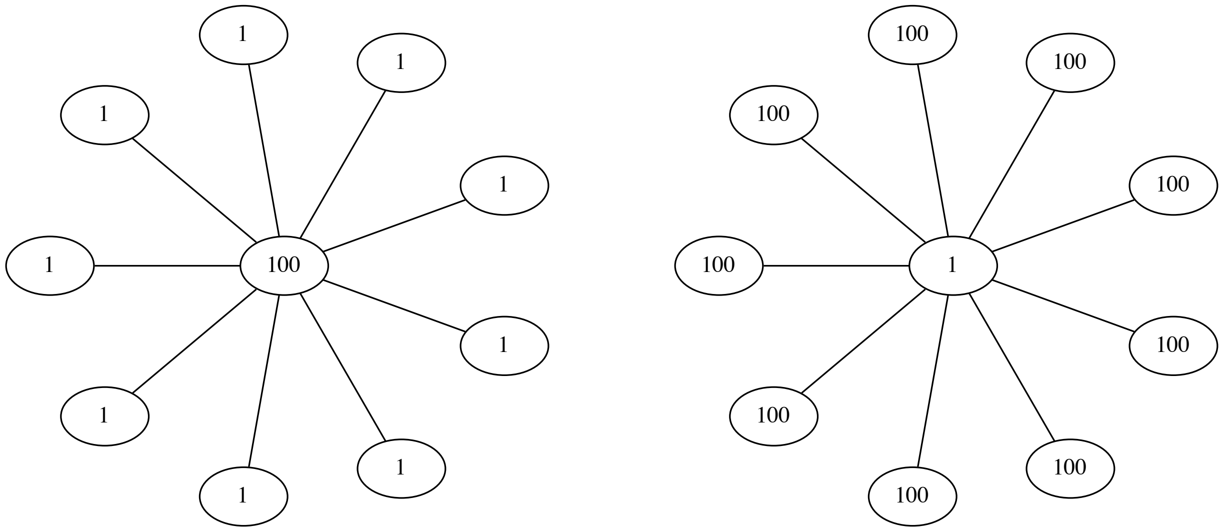


Figure 1: These are two examples of the MIN-WEIGHT-VERTEX-COVER problem, where the value in each vertex represents the weight. Notice in the first case, the optimal vertex cover includes all the leaves; in the second case, the optimal vertex cover includes only the center vertex. In both cases, the simple (deterministic) 2-approximate vertex cover algorithm fails badly, always including a vertex of cost 100.

3.2 MIN-WEIGHT-VERTEX-COVER as an Integer Linear Program

Assume we are given a graph $G = (V, E)$ with weight function $w : V \to \mathbb{R}$, and we want to find a minimum weight vertex cover.

The first step is to define a set of variables. In this case, it is natural to define one variable for each vertex in the graph. Assuming there are n vertices in the graph[9], we define variables $x_1, x_2, \ldots, x_n$. These variables should be interpreted as follows: $x_j = 1$ implies that vertex v_j is included in the vertex cover; $x_j = 0$ implies that vertex v_j is *not* included in the vertex cover.

Given these variables, we can now define the objective function. We want to minimize the sum of the weights of the vertices included in the vertex cover. That is, we want to minimize: $\sum_{j=1}^{n} w(v_j) \cdot x_j$.

Finally, we define the constraints. First, we need to ensure that every edge is covered: For every edge $e = (v_i, v_j)$, we need to ensure that either v_i or v_j is in the vertex cover. We need to represent this constraint as a linear function. Here is one way to do that: $x_i + x_j \geq 1$. This ensures that either x_i or x_j is included in the vertex cover.

We need one more constraint: for each variable x_j, we need to ensure that either $x_j = 1$ or $x_j = 0$. What would it mean if the Linear Programming solver returned that $x_j = 0.4$? This would not be useful. Unfortunately, there is no good way to represent this constraint as a linear function.

And it should not be surprising to you that we cannot represent the MIN-VERTEX-COVER problem as a Linear Program! We know, already, that MIN-VERTEX-COVER is NP-hard. We also know that Linear Programs can be solved in polynomial time. Thus, if we found a Linear Programming formulation for MIN-VERTEX-COVER, that would imply that $P = NP$.

[9] As n can go beyond 3 (variables/dimensions), it is rather impossible to properly visualize the n-dimensional Linear Program of this MIN-WEIGHT-VERTEX-COVER problem.

Instead, we formulate an *Integer* Linear Program:

Definition 3 *An **Integer Linear Program** (ILP) is a Linear Program in which all the variables are constrained to be integer values.*

Thus, we can now represent MIN-WEIGHT-VERTEX-COVER as an Integer Linear Program as follows:

$$\min\left(\sum_{j=1}^{n} w(v_j)\cdot x_j\right) \qquad \text{where:}$$

$$\begin{array}{rcll} x_i + x_j & \geq & 1 & \text{for all } (i,j)\in E \\ x_j & \geq & 0 & \text{for all } j \in V \\ x_j & \leq & 1 & \text{for all } j \in V \\ x_j & \in & \mathbb{Z} & \text{for all } j \in V \end{array}$$

Notice that the objective function is a linear function of the variables x_j, where the weights are simply constants. (There is no term in the expression that looks like $x_i x_j$, i.e., multiplying variables together nor term that looks like x^y for $y \neq 1$.) Similarly, each of the constraints is a linear function of the variables. The only constraint that cannot be expressed as a linear function is the last one, where we assert that each of the variables must be integral. (We will often abbreviate the last three lines by simply stating that $x_j \in \{0,1\}$.)

3.3 Relaxation

Unfortunately, there is no polynomial time algorithm for solving ILPs. We have already effectively shown that solving ILPs is NP-hard. If there were a polynomial time algorithm, we would have proved that $P = NP$. Instead, we will *relax* the ILP to a (regular) LP. That is, we will consider the same optimization problem, but dropping the constraint that the variables be integers. Here, for example, is the MIN-WEIGHT-VERTEX-COVER **relaxation**:

$$\min\left(\sum_{j=1}^{n} w(v_j)\cdot x_j\right) \qquad \text{where:}$$

$$\begin{array}{rcll} x_i + x_j & \geq & 1 & \text{for all } (i,j)\in E \\ x_j & \geq & 0 & \text{for all } j \in V \\ x_j & \leq & 1 & \text{for all } j \in V \end{array}$$

Notice that the solution to this LP is no longer guaranteed to be a solution to MIN-WEIGHT-VERTEX-COVER! In fact, there is no obvious way to interpret the solution to this LP. What does it mean if we decide that $x_j = 0.4$?

Solving the relaxed ILP does tell us something: The solution to the Linear Program is *at least as good* as the optimal solution for the original ILP. In the case of MIN-WEIGHT-VERTEX-COVER, imagine that we solve the relaxed ILP and the LP solver returns a set of variables $x_1, x_2, \ldots, x_n$ such that $\left(\sum_{j=1}^{n} w(v_j)\cdot x_j\right) = c$, for some value c. Then we know that $OPT(G) \geq c$, where $OPT(G)$ is the optimal (integral) solution for MIN-WEIGHT-VERTEX-COVER.

Why? Imagine there were a better solution $x'_1, x'_2, \ldots, x'_n$ return by $OPT(G)$. In that case, this solution would also be a feasible solution for the relaxed Linear Program: Each of these variables x'_j is either 0 or 1, and hence would be a valid choice for the relaxed case where $0 \leq x_j \leq 1$. Hence the LP solver would have found this better solution.

The general rule is that when you expand the space being optimized over, your solution can only improve. By relaxing an ILP, we are expanding the range of possible solutions, and hence we can only find a better (or at least equal) solution.

Lemma 4 *Let I be an Integer Linear Program, and let $L = relax(I)$ be the relaxed Integer Linear Program. Then $OPT(I) \geq OPT(L)$.*

3.4 Solving MIN-WEIGHT-VERTEX-COVER

Returning to MIN-WEIGHT-VERTEX-COVER, we have defined an ILP I for solving MIN-WEIGHT-VERTEX-COVER. We have relaxed this ILP and generated an LP L. The first thing we must argue is that there is a feasible solution the Linear Program:

Lemma 5 *The relaxed ILP for the vertex cover problem has a feasible solution.*

Proof Consider the solution where each $x_j = 1$. This solution satisfies all the constraints. □

Assume we have now solved the LP L using an LP solver and discovered a solution $x_1, x_2, \ldots, x_n$ to the Linear Program L. Our goal is to use this (non-integral) solution to find an (integral) solution to the MIN-WEIGHT-VERTEX-COVER problem.

Here is a simple observation: If (u, v) is an edge in the graph, then either $x_u \geq 1/2$ or $x_v \geq 1/2$. Why? Well, the Linear Program guarantees that $x_u + x_v \geq 1$. The Linear Program may well choose non-integral values for x_u and x_v, but it will always ensure that all the (linear) constraints are met.

Consider, then, the following procedure for ***rounding*** our solution to the Linear Program:

- For every vertex $u \in V$: if $x_u \geq 1/2$, then add u to the vertex cover C.

We claim that the resulting set C is a vertex cover:

Lemma 6 *The set C constructed by rounding the variables $x_1, x_2, \ldots, x_n$ is a vertex cover.*

Proof Assume, for the sake of contradiction, that there is some edge (u, v) that is not covered by the set C. Since neither u nor v was added to the vertex cover, this implies that $x_u < 1/2$ and $x_v < 1/2$. In that case, $x_u + x_v < 1$, which violates the constraint for the LP, which is a contradiction. □

It remains to show that the rounded solution is a good approximation, i.e., that we have not increased the cost too much[10].

Lemma 7 *Let C be the set constructed by rounding the variables $x_1, x_2, \ldots, x_n$. Then $cost(C) \leq 2 \times cost(OPT)$.*

Proof The proof relies on two inequalities. First, we relate the cost of OPT to the cost of the Linear Program solution:

[10] Unless, of course, being up to two times worse than optimal, a.k.a. 200%-off from optimal, is not an acceptable criteria.

$$cost(OPT) \geq \sum_{j=1}^{n} w(v_j) \cdot x_j \tag{1}$$

This follows because the x_j were calculated as the optimal solution to a relaxation of the original MIN-WEIGHT-VERTEX-COVER problem. Recall, by relaxing the ILP to an LP, we can only improve the solution (i.e., in this case, we can only get a solution that is $\leq$ the integral solution).

Second, we related the cost of the LP solution to the cost of the rounded solution. To represent the rounded solution, let $y_j = 1$ if $x_j \geq 1/2$, and let $y_j = 0$ otherwise. Now, the cost of the final solution, i.e., $cost(C)$, is equal to $\sum_{j=1}^{n} w(v_j) \cdot y_j$. Notice, however, that $y_j \leq 2x_j$, for all j. Therefore:

$$\begin{aligned}
\sum_{j=1}^{n} w(v_j) \cdot y_j &\leq \sum_{j=1}^{n} w(v_j) \cdot (2x_j) \\
&\leq 2\left(\sum_{j=1}^{n} w(v_j) \cdot x_j\right) \\
&\leq 2 \times OPT(G)
\end{aligned}$$

Thus, the rounded solution has cost at most $2 \times OPT(G)$. □

3.5 General Approach

We have thus discovered a polynomial time 2-approximation algorithm for the MIN-WEIGHT-VERTEX-COVER:

- Define the MIN-WEIGHT-VERTEX-COVER problem as an Integer Linear Program (ILP).
- Relax the ILP to a standard Linear Program (LP).
- Solve the LP using an existing LP solver of your choice.
- Round the solution, adding vertex v to the cover if and only if $x_j \geq 1/2$.

The general approach we have defined here for MIN-WEIGHT-VERTEX-COVER can also be applicable to a *wide variety* of Combinatorial Optimization Problems (COPs). The basic idea is to find an ILP formulation, relax it to an LP, solve the LP, and then round the solution until it is integral. MIN-WEIGHT-VERTEX-COVER yields a solution that is particularly easy to round. For other problems, however, rounding the solution may be more difficult.

One question, perhaps, is when this approach works and when this approach fails. Sometimes, the non-integral solution (returned by the LP solver) will be much better than the optimal integral solution. In such a case, it will be very difficult to round the solution to find a good approximation. This ratio is defined as the ***integrality gap***. Assume I is some Integer Linear Program:

$$\textbf{integrality gap} = \frac{OPT(relax(I))}{OPT(I)} \tag{2}$$

For a specific problem, if the integrality gap is at least c, then the best approximation algorithm you can achieve by rounding the solution the relaxed Integer Linear Program is a c-approximation. This is a fundamental limit on this technique for developing approximation algorithms.

4 Linear Programming Terminologies

In this section, we will review some basic Linear Programming terminologies.

4.1 Definitions

In general, a Linear Program consists of:

- A set of variables: $x_1, x_2, \ldots, x_n$.
- A linear objective to maximize (or minimize): $c_1x_1 + c_2x_2 + \cdots + c_nx_n$. Thinking of c and x as vectors, this is often written more tersely as: $c^T x$ (where c^T represents the transpose of c, and multiplication here represents the dot product.)
- A set of linear constraints of the form: $a_{j,1}x_1 + a_{j,2}x_2 + \cdots + a_{j,n}x_n \leq b_j$. (This version represents the jth constraint.) This is often abbreviated by the matrix equation: $Ax \leq b$.

Putting these pieces together, a Linear Program is often presented in the following form:

$$\begin{array}{rcl} \max cx & & \text{where} \\ Ax & \leq & b \\ x & \geq & 0 \end{array}$$

4.2 Terminology

Standard terminologies for Linear Programs:

- A point x is ***feasible*** if it satisfies all the constraints.
- An LP is bounded if there is some value V such that $c^T x \leq V$ for all points x.
- Given a point x and a constraint $a^T x \leq b$, we say that the constraint is ***tight*** if $a^T x = b$; we say that the constraint is ***slack*** if $a^t x < b$.
- A ***halfspace*** is the set of points x that satisfy one constraint. For example, the constraint $a^t x \leq b$ defines a halfspace containing all the points x for which this inequality is true. A halfspace is a convex set.
- The ***polytope*** of the LP is the set of points that satisfy all the constraints, i.e., the intersection of all the constraints. The polytope of an LP is convex, since it is the intersection of halfspaces (which are convex).
- A point x is a vertex for an n-dimensional LP if there are n linearly independent constraints for which it is tight.

For every Linear Program, we know that one of the following three cases holds:

- The LP is infeasible. There is no value of x that satisfies the constraints.
- The LP has an optimal solution.
- The LP is unbounded.

(Mathematically, this follows from the fact that if the LP is feasible and bounded, then it is a closed and bounded subset of $\mathbb{R}^n$ and hence has a maximum point.)

4.3 Standard Form

In general, we will think of Linear Programs as having a standard (or canonical) form (see Section 29.1 of [1], omitting the slack form for now). Many ready made LP solvers out there assume this standard form.

For a maximization problem, the standard form is:

$$\begin{aligned} \max cx & \quad & \text{where} \\ Ax & \leq & b \\ x & \geq & 0 \end{aligned}$$

For a minimization problem, the standard form is:

$$\begin{aligned} \min cx & \quad & \text{where} \\ Ax & \geq & b \\ x & \geq & 0 \end{aligned}$$

Notice that the standard forms obey the following rules:

- Inequalities: Every constraint is an inequality. For a maximization problem, the inequalities are all $\leq$ (except for the $x \geq 0$ constraints). For a minimization problem, the inequalities are all $\geq$.
- Non-negative: Every variable is constrained to be ≤ 0.

(Beware in other classes or other textbooks, there may be different preferred "standard forms".)

It is easy to translate Linear Programs in other forms into standard form. For example, we can translate a minimization problem into a maximization problem (and vice versa) by negating the objective function. We can reverse the direction of an inequality by negating the constraint. We can translate an equality into an inequality by introducing two constraints. Etc.

Imagine, for example, that you are given the following Linear Program and your LP solver is designed to solve a maximization problem:

$$\begin{aligned} \min x_1 + 2x_2 - x_3 & \quad & \text{where // this is a minimization problem} \\ x_1 + x_2 & = & 7 \text{ // this is an equality} \\ x_2 - 2x_3 & \geq & 4 \text{ // this is a } \geq \text{ inequality} \\ x_1 & \leq & 2 \end{aligned}$$

First, we can translate this into a maximization problem by negating the objective function:

$$\begin{aligned} \max -(x_1 + 2x_2 - x_3) & \quad & \text{where} \\ x_1 + x_2 & = & 7 \text{ // this is an equality} \\ x_2 - 2x_3 & \geq & 4 \text{ // this is a } \geq \text{ inequality} \\ x_1 & \leq & 2 \end{aligned}$$

Then, we can replace the equality with two inequalities:

$$\begin{aligned}
\max -(x_1 + 2x_2 - x_3) & \quad & \text{where} \\
x_1 + x_2 & \leq & 7 \\
x_1 + x_2 & \geq & 7 \text{ // this is a } \geq \text{ inequality} \\
x_2 - 2x_3 & \geq & 4 \text{ // this is a } \geq \text{ inequality} \\
x_1 & \leq & 2
\end{aligned}$$

Next, we can flip the direction on the inequalities that are in the incorrect direction:

$$\begin{aligned}
\max -(x_1 + 2x_2 - x_3) & \quad & \text{where} \\
x_1 + x_2 & \leq & 7 \\
-(x_1 + x_2) & \leq & -7 \\
-(x_2 - 2x_3) & \leq & -4 \\
x_1 & \leq & 2
\end{aligned}$$

The last step in putting the LP in standard form is to constrain all the variables to be positive. To accomplish this, we create new variables to replace the variables here. Specifically, we define variables $x_1^+, x_1^-, x_2^+, x_2^-, x_3^+, x_3^-$. We define them as follows:

$$\begin{aligned}
x_1 &= (x_1^+ - x_1^-) \\
x_2 &= (x_2^+ - x_2^-) \\
x_3 &= (x_3^+ - x_3^-)
\end{aligned}$$

We can then constrain the new variables to all be positive. We end up with the following LP in standard form:

$$\begin{aligned}
\max -((x_1^+ - x_1^-) + 2(x_2^+ - x_2^-) - (x_3^+ - x_3^-)) & \quad & \text{where} \\
(x_1^+ - x_1^-) + (x_2^+ - x_2^-) & \leq & 7 \\
-((x_1^+ - x_1^-) + (x_2^+ - x_2^-)) & \leq & -7 \\
-((x_2^+ - x_2^-) - 2(x_3^+ - x_3^-)) & \leq & -4 \\
(x_1^+ - x_1^-) & \leq & 2 \\
x_1^+ & \geq & 0 \\
x_1^- & \geq & 0 \\
x_2^+ & \geq & 0 \\
x_2^- & \geq & 0 \\
x_3^+ & \geq & 0 \\
x_3^- & \geq & 0
\end{aligned}$$

Solving this final LP (and translating back to the initial variables) gives us the correct optimal solution for the original LP. (Notice that it may increase the number of variables and constraints by a constant factor).

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 3rd edition, 2009.

Index