Computability, Complexity, and Algorithms

Charles Brubaker and Lance Fortnow

Linear Programming - (Udacity)

Introduction - (Udacity, Youtube)

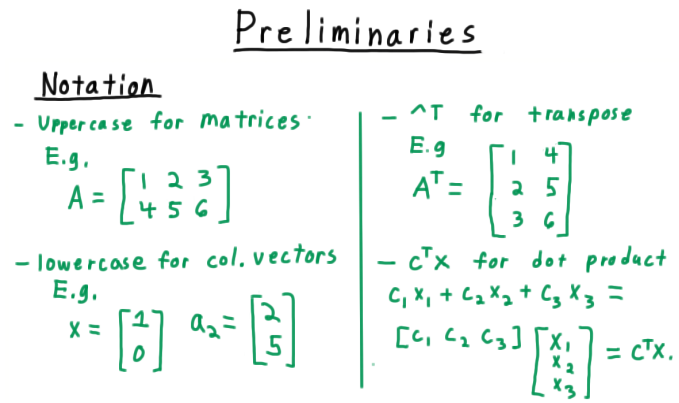
The subject for this lesson is linear programming. We've seen some very general tools and abstractions in this course, but it would be hard to argue that any other combines the virtues of simplicity, generality and practicality as well as linear programming. It is simple enough to be captured with just a few matrix expressions and inequalities, general enough so that it can be used to solve any problem in P in polynomial time , and practical enough that it helped revolutionize business and industry in the middle of the twentieth century. In many ways, this is algorithms at its best.

Preliminaries - (Udacity, Youtube)

This lesson begins by reviewing the two dimensional linear programming problems that high school students often solve in their Algebra 2 classes. Then, it extends the the equations to N dimensions and captures the essential intution–that optimal solutions are at the corners of the allowed region– with the Fundamental Theorem of Linear Programming. Finally, it covers the simplex algorithm, a very practical way for solving these optimizations.

There are many good references for linear programming. This treatment will follow most closely David Luenberger's, *Linear and Nonlinear Programming.*

Before we begin, however, I should mention that parts of this lecture will use notation and some ideas from linear algebra. Some notation we will use is summarized in the image below.



As far as concepts go, the ideas of how to represent systems of equations as matrices, of linear indepedence of vectors, matrix rank, and the inverse of a matrix should all be familiar. If they aren't, then it would be a good idea to refresh your understanding before watching the rest of this lesson.

As always, it is recommended that you watch with pencil and paper handy so that you can pause the video and work out details on your own as needed.

HS Linear Programming - (Udacity, Youtube)

I want to begin our discussion of linear programming with a kind of problem that you likely first encountered in a High School algebra class. A graduate student is trying to balance research and relaxation time. He figures that eating, sleeping, and commuting leave him with 14 hours in the day for other activities. He has also found that after two hours of work, he needs to relax for a half-hour before he can work effectively for another hour again. Of course, his advisor wants him to work as much as possible.

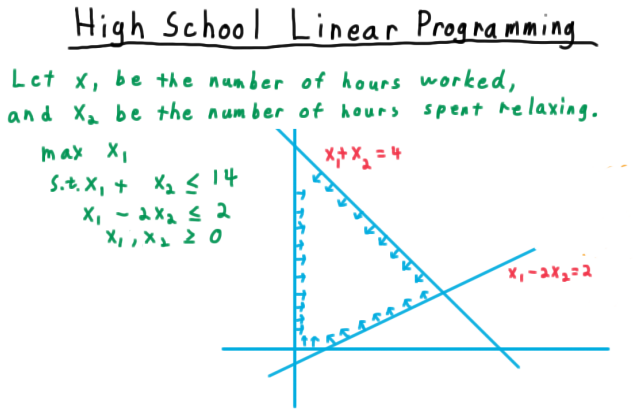
We'll let x_1 be the amount of time spent on research and x_2 the amount of time spent relaxing. Then we can express the graduate student's time management problem as the following optimization.

$$\begin{array}{lrcrcr} \max & x_1 & & & & \\ \text{s.t.} & x_1 & + & x_2 & \leq & 14 \\ & x_1 & - & 2x_2 & \leq & 2 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

We express the fact that he only has 14 hours for these activities by saying that $x_1 + x_2$ is at most 14. We express the fact that he feels for half as much relaxations as work after two hours of work with the second constraint $x_1 - 2x_2 \leq 2$. Of course, he can't spend negative time on either of these activities, so we need to add that contraint as well. The overall goal is to maximize time worked, so we make that our objective function, and we

want to maximize that subject to these contraints.

Now in HS, your teacher probably asked you to begin by graphing the inequalities. When we do this, we see that the constraints generate the following polytope.

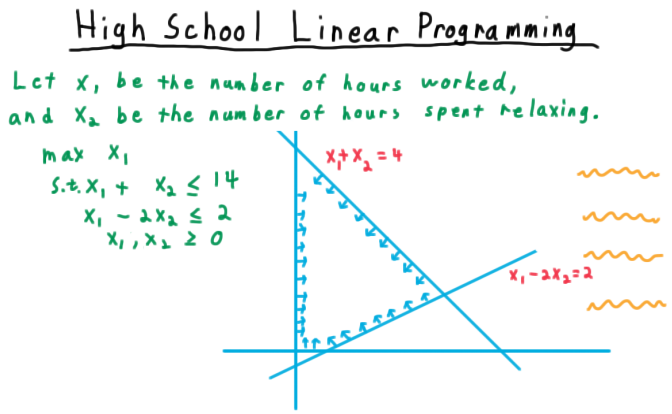


Perhaps, your HS teacher didn't use the word polytope that's what this region here is. Each constraint restricts our solution to half of the plane, called a half-space, and a polytope is the intersection of half-spaces.

After you graph this region, the solution can be picked out as one of the the vertices. In this case, it's pretty easy to see that it's this one on the right, which is at the intersection of the two problem constraints. Maybe, if the formula was a little more complicated and you weren't sure, you could have tested each one of the vertices and picked the one with the highest objective value.

Why is the optimal solution at one of the vertices? Well, remember that in this problem and all similar ones from High school, the objective function, the thing we're optimizing, is *linear*. The only thing that matters is how far we can move in a certain direction , in this case the x_1 direction, but it could be any direction in the plane.

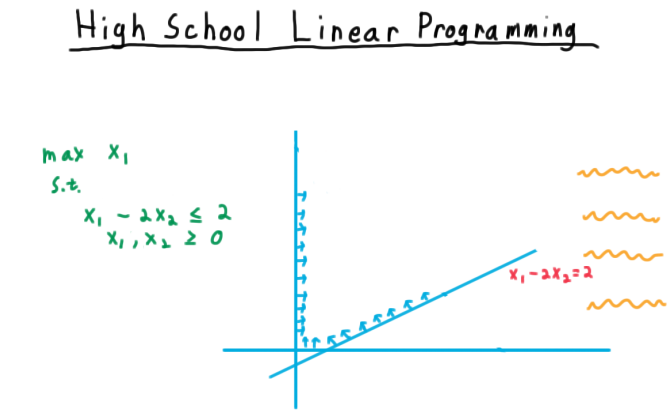
If you like, you can think of there being a giant magnet infinitely far away pulling our point x in a certain direction.



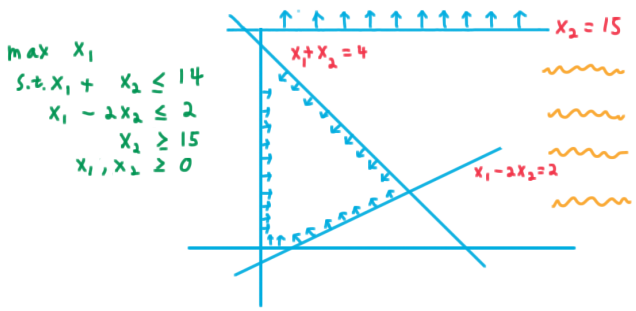
In trying to get as close as possible to the magnet, this point must end up at one of the vertices.

If some point is interior, then we can clearly improve by moving in this direction. If a point is on an edge, then we can improve by moving along this edge. The only time we couldn't improve in this way would be if the edge were perpendicular to the direction we wanted to move in. But then \emph{both} vertices on either side of the segment have the same value and therefore are also optimal solutions.

Thinking more abstractly, there isn't always an optimal solution, as there is in this particular case. If I eliminate one contraint as shown below, then polytope is unbounded in the gradient direction for our objective.



In this case, we can keep moving our point x further and further getting greater values for our object. You give me an x–I've got a better one! Hence there is no optimal solution. On the other hand, if I put back that constraint and add another one, we might find that they are contradictory.



There is no way to satisfy them. If there are no solutions, there can't be an optimal one.

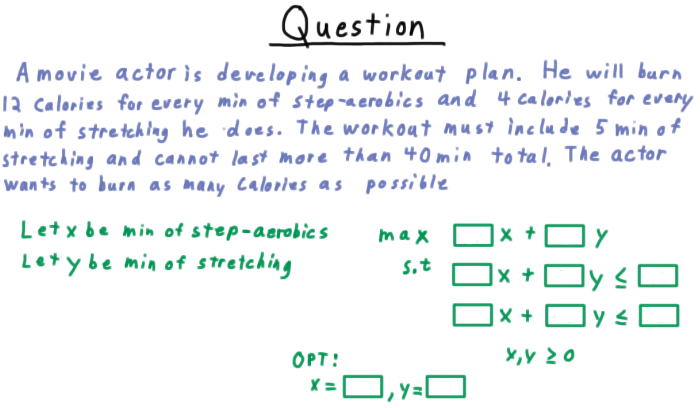
So those are the three things that can happen: the contraints can create a bounded region and we find an optimum, the region can be unbounded, in which case we might find an optimum or the problem might be unbounded, or the region can be empty.

Workout Plan - (Udacity)

Let's do a quick exercise on this High School linear programming. Here's the problem:

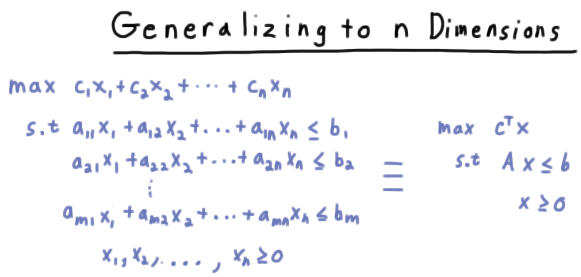
A movie actor is developing a workout plan. He will burn 12 Calories for every minute of step-aerobics he does and 4 for every minute of stretching he does. The workout must include 5 minutes of stretching and must last no longer than 40 minutes in total. The actor wants to burn as many Calories as possible.

We'll let x be the number of minutes spent on step-aerobics and let y be the number of minutes spent stretching.
I want you to express the actor's problem as a linear program and give the optimal values for x and y in the boxes below.



To n Dimensions - (Udacity, Youtube)

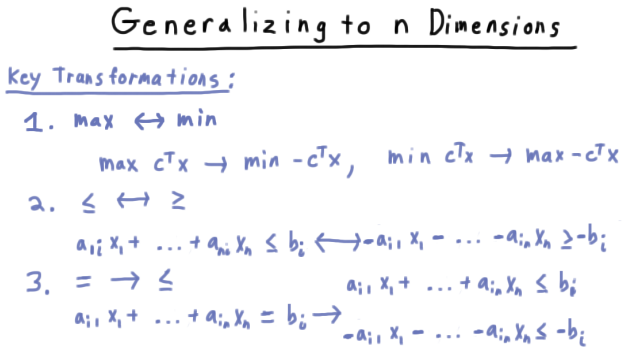
Linear Programming is largely just the generalization of this sort of problem solving to n dimensions, instead of just the two that we've used so far.



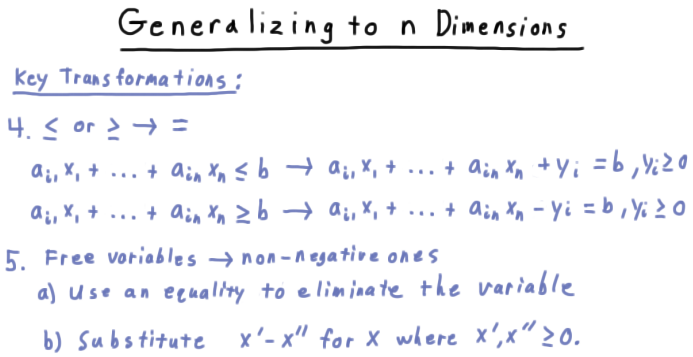
(Note that inequality over matrices means that the inequality holds for each element.)

When we first encounter an linear programming optimization problem, it might not be in this form. In fact, the only requirements for an optimization problem being a linear program are that both the objective function and the constraints, inequalities or equalities, be *linear*. If this is true then, we can always turn it into a canonical form like this one.

Here are the key transformations.

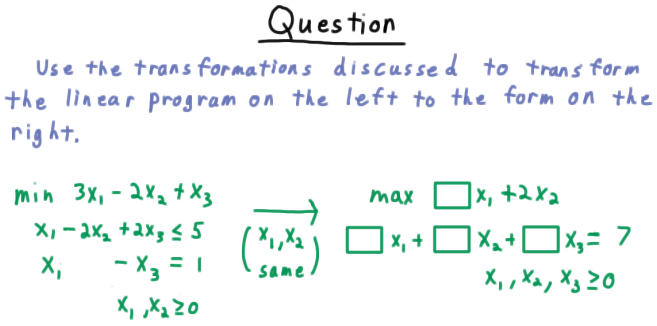


Things get a little more interesting when we go from one of the inequalites to an equality. Here we introduce a new variable, called a slack or surplus variable depending on the inequality.
There is also the problem of free variables that are allowed to be negative. There are two ways to cope with one of these. If it is involved in an equality constraint, then you can often simply eliminate it through substitution. Otherwise, you can replace it with the difference of two *new* non-negative variables.



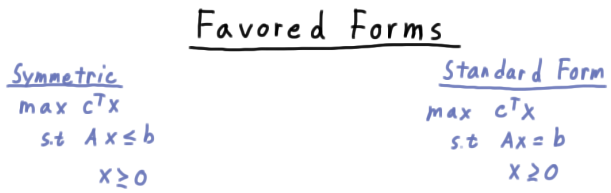
Transformation Quiz - (Udacity)

Let's do a quick exercise to practice these transformations.



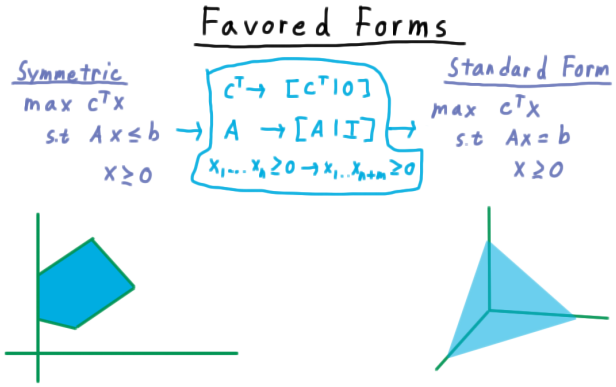
Favored Forms - (Udacity, Youtube)

By these various transformations, it's possible to write any linear program in a variety of forms. Two forms, however, tend to be the most convenient and widely used. First is what we'll call the **symmetric form**–we'll see why it gets that name when we consider duality– and second, is the **standard form.** The key difference between the two is that we have changed the inequality for an equality.



To better understand the relationship between these two forms, I'm going to write the standard form in terms of the symmetric form.

To convert the m inequalitys to equalities we introduce m slack variables $x_{n+1} \ldots x_{n+m}$. Of course, this means that we need to augment our matrix A as well so that these slack variables can do their job. And c also needs to be augmented so that we can multiply it with the new x without changing the value of the objective function.



Geometrically, we've switched our optimization from being over a polytope in n dimensions (note the inequalities in the symmetric form) to being over the intersection of a flat (note the equality constraints) intersected with cone defined by the positive coordinate axes (note the non-negativity constraints) in $n + m$ dimensions.

We expect that an optimum for the symmetric problem will be one of the vertices of the polytope, where n of the hyperplanes defined by the constraints intersect. That is to say, of these $n + m$ constraints, (m from A and n from the non-negativity of x) n must hold with equality, or be "tight" in the common parlance. Some might come from A, others from the non-negativity constraints, but there will always be n tight constraints.

Over in standard form, the notion of whether the contraints from A are tight are not is captured by the slack variables that we introduced. A slack variable is zero if and only if the corresponding contraint is tight. Thus, at least n of the variables will be zero when we are at a vertex of the original polytope. In fact, if I tell you which n variables are zero and these correspond to an linearly independent set of constraints, then you can construct the rest of the variables based on the equality constraints.

Now, so far I have kept on using the number of variables n and the number of contraints m from the symmetric form, even as we talk about the standard form. In general, however, when discussing the standard form we redefine the new n to be the total number of variables (the old $n + m$).

One other thing to note about this equality form is that enforce the matrix A to have rank m where m is the number of constraints. That is to say, the rows should be linearly independent. If the rows aren't linearly indepedent, there are two possibilities. One is that the constraints are inconsistent meaning that there is no solution. The other possibility is that the constraints are redundant meaning that some of them can be eliminated. So from now on we'll assume that A has full rank.

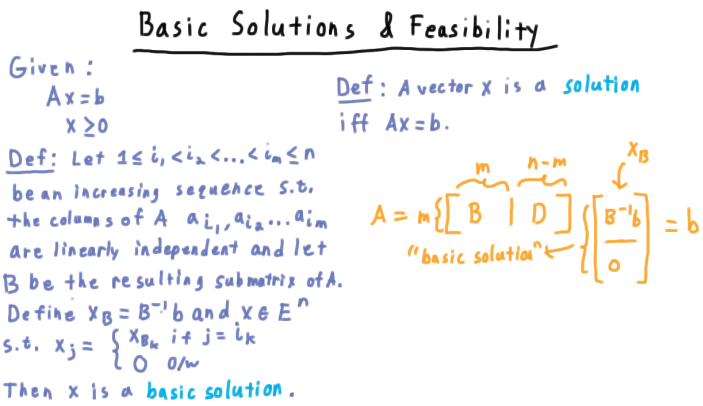
Basic Solutions and Feasibility - (Udacity, Youtube)

From this point onward, we will consider linear programs in the standard form, where the constraints are expressed as equalities. We will have an underdetermined system of equations that has lots of solutions, and we will try to find the one that maximizes the objective function. Now, if you have ever had to come up with a solution to an underdetermined system on your own, you will have noticed that it's easiest to find one by simply setting some of the coefficients to zero so as to create a square system and then solving for the rest. In effect, this is what solvers like Matlab do as well. These solutions, are called basic solutions, and it turns out that to solve linear programs basic solutions are the only ones that we will need to consider. The trick, however, is figuring out which coefficients need to set to zero. With this intuition in mind, let's dive into the details.
I want to define some vocabulary that will be useful going forward.
First, we say that a vector x is a **solution** if it solves

$$Ax = b.$$

A **basic solution** is one generated as follows: we'll pick an increasing sequence of m column numbers so that the corresponding columns are independent and call the resulting matrix B. This is easiest to see when the chosen columns are the first m, and we'll use this convention for most of our treatment.
We define $x_B = B^{-1}b$ and then embed this in the longer vector x, putting in the value from x_B for columns in our sequence and zero otherwise. Then x is a basic solution.

Really, all that we're trying to accomplish here is to let x_B get multiplied with the columns of A corresponding to B and have zero multiplied with all the other columns. (Remember post-multiplication corresponds to column operations).



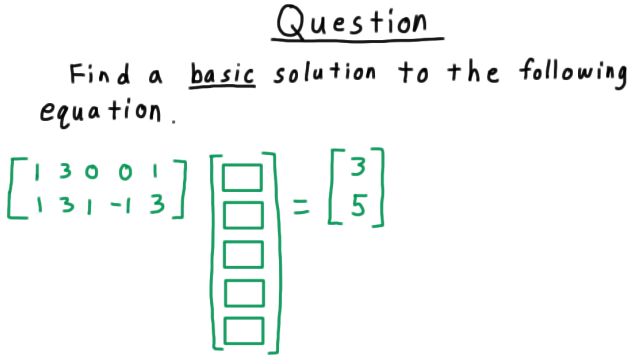
So that's a basic solution, and we call it basic because it came from our choice of this linearly independent set of columns, which forms a basis for the column space. From the basis, we get a basic solution.

It is possible for more than one basis to yield the same basic solution if some of the entries of x_B are zero. Such a solution is called **degenerate.** This corresponds to a vertex being the intersection of more than n hyperplanes in the symmetric form.

So far, this vocabularly only addresses the equality constraints. Adding in the non-negativity constraints on the variables, we use the word "feasible." Thus, a **feasible solution** is a solution that has all non-negative entries, and a **basic feasible solution** is one that comes from a basis as described above and has all non-negative entries.

Find a Basic Solution - (Udacity)

Now for an exercise on basic solutions: Given the equations below, find a basic solution for x. Remember that since there are only two rows in the matrix your solution most not have more than two non-zero entries.



Fundamental Theorem of LP - (Udacity, Youtube)

So far, we've reminded ourselves of the basics of linear programming by examining it in two dimensions. Then we built up some vocabulary and notation that allow us to extend these notions to n dimensions. Now, we are ready for the culmination of all this work in the fundamental theorem of linear programming, which captures the idea that the optimal solutions should be a the corners (also called extremes points) of the feasible region, and tells us that we need only consider basic feasibles solutions.

Fundamental Theorem of LP

Given a linear program in the form

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

where A is an $m \times n$ matrix of rank m

i) if there is a feasible solution, there is a basic feasible solution

ii) if there is an optimal feasible solution, there is an optimal basic feasible solution.

We'll start by proving the first point of the theorem statement above.
Let x be a feasible solution and we'll consider only the positive entries. Without loss of generality, let's assume that they are the first p. Then it must be the case that that

$$x_1 a_1 + \ldots + x_p a_p = b.$$

That is, after all, part of what it means to be feasible.

Case 1: Suppose first that the columns $a_1 \ldots a_p$ are linearly independent.

Then, it's not possible that p should be greater than m. If $p = m$, then x is basic, and we're done. The quanity p could be less than m, but then we would just add columns as needed until we formed a basis. That covers the independent case.

Case 2: Suppose that $a_1 \ldots a_p$ are linear dependent.

That means that there are coefficients $y_1 \ldots y_p$ such

$$y_1 a_1 + \ldots + y_p a_p = 0$$

with at least one of these coefficients being positive. We'll then choose

$$\epsilon = \min\{x_i / y_i | y_i > 0\}.$$

Then multiplying the above equation in y by ϵ and subtracting it from $x_1 a_1 + \ldots x_p a_p = b$, we end up with another feasible solution. This one, however, has at most $p - 1$ positive coefficients because our choice of ϵ sent at least one of them to zero. We can then repeat this argument as needed to reduce the problem to case 1.

Now, onto part 2 of the theorem, which shows that if there is an optimal feasible solution, there is an optimal basic feasible solution. This will feel similar to the first part. We let x be an optimal feasible solution, meaning that not only is it a solution but it also has the highest possible dot-product with c.
As before, we'll let columns $a_1 \ldots a_p$ correspond to the non-zero entries of x and consider first the case where these are linearly independent. The situation is the same as before. p being greater than m is impossible, equal means that it is a basic solution, and less just means that it is a denerate soluiton.
This case is simple.

If the columns are *dependent,* then we have a set of coefficients $y_1 \ldots y_p$ with at least one positive so that

$$y_1 a_1 \ldots y_p a_p = 0.$$

Note, however, that for ϵ sufficiently close to zero (both positive and negative) $x - \epsilon y$ is feasible. Thus, $c^T y = 0$. Otherwise, we could choose the sign of ϵ so as to make

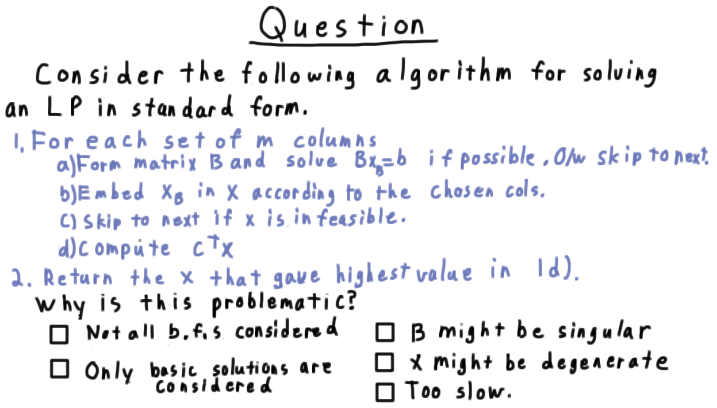
$$c^T x < c^T (x - \epsilon y).$$

Since we assumed that x is an optimal solution we conclude that $c^T y = 0$. Therefore, we can set ϵ to the same choice as before that sent one of the coefficients $1 \ldots p$ to zero. By repeating this argument, we eventually reach case 1.

We've just seen how the fundamental theorem of linear programming tells us that we can alway achieve an optimal value for the program with a basic solution. Moreover, basic solutions come from a choice of m linearly independent columns for the basis. Remember this key point going forward.

Brute Force Algorithm - (Udacity)

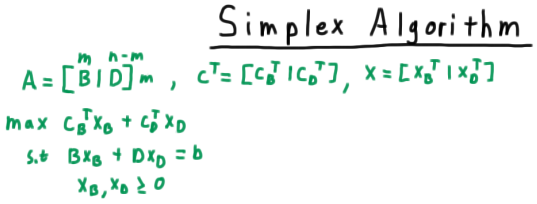
The fundamental theorem of linear programming immediately suggests an algorithm where we just try all the possible bases and take the best generated basic feasible solution, as outlined here. And my question for you is "Why is this algorithm problematic or impractical?" Check all that apply.



Simplex Equations - (Udacity, Youtube)

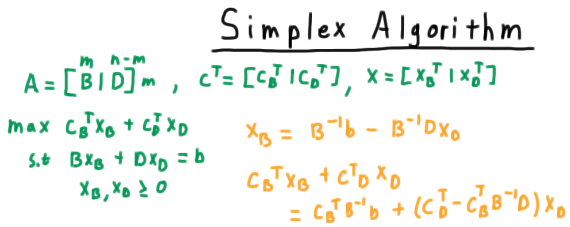
Now, we'll talk about a much more efficient approach called the Simplex Algorithm. This actually will not be polynomial time in the worst case, but as a practical matter this algorithm is efficient enough to be widely used. The Simplex algorithm allow us to move from one basic feasible solution to a better, and by moving to better and better basic feasible solutions, we eventually reach the optimum.

For convenience, let's suppose that our current basis consists of the first m columns of A. We'll call this submatrix B and the remaining submatrix D. It will also be convenient to partition x and c in an analogous fashion. Overall, then we can rewrite our standard form as



For the simplex algorithm, we want to consider the effects of swapping out one of the current basis columns for another one. To do this, we first want to identify a good candidate for moving into the basis, one that will improve the objective function. As the program stands, however, it not immediately which if any are good candiates. Sure, for some x_i the coefficient might be positive, but raising that value might force others to change because of the contraints, making the whole picture rather opaque. Therefore, it will be convenient to parameterize our ability to move around in the flat defined by the equality constraints solely in terms of x_D, the variables that we are thinking about moving into the basis.

To this end, we solve the equality constraint for x_B so that we can substitute for it where desired. First we substitute it into the objective function, and through a little manipulation, we get this expression.



The constant term here doesn't matter, since we are only considering the effects of changing x_D. This quantity here that is multiplied with x_D, however, is important enough that it deserves its own name. Let's call it r_D.

$$r_D^T = c_D^T - c_B^T B^{-1} D$$

In our reframing of the problem, therefore, we want to maximize $r_D^T x$. How about the other constraints? Well, the first one goes away with the substitution. The requirement, however, that x_B remain non-negative remains.
Substituting our equation for x_B, we get the linear program

$$\begin{aligned} \max \quad & r_D^T x_D \\ \text{s.t.} \quad & B^{-1} D x_D \leq B^{-1} b \\ & x_D \geq 0 \end{aligned}$$

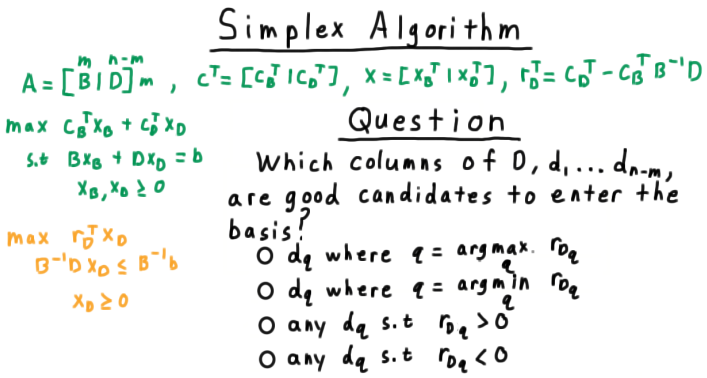
where, of course, x_D must remain non-negative as well.

Note $x_D = 0$ the current situation for the algorithm is feasible, and it is very easy to see a way to improve the objective value just by looking at the vector r_D. Our real goal, however, is not just to climb uphill but to figure out which column should enter the basis.

Who Enters the Basis - (Udacity)

Actually then, we'll make this quiz! What makes for a good candidate to enter the basis?

Check the best answer.



The answer is that any column corresponding to a positive entry of r_D is a good candidate. We want the entry to be positive because the corresponding element of x_D is also going to positive as we increase it. Just picking the greatest entry of r_D doesn't work because this still might be negative.

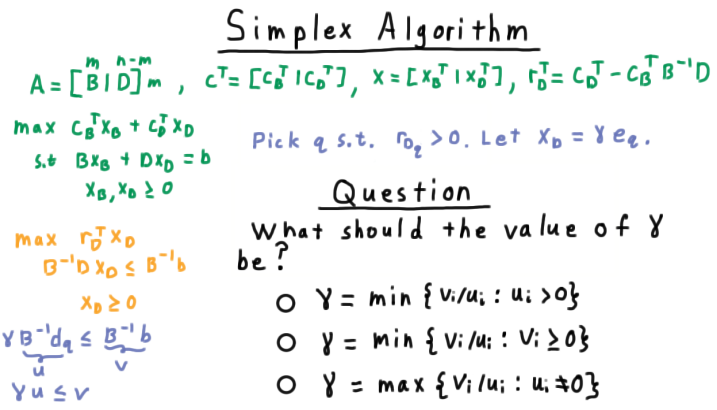
This idea then becomes the basis for the simplex algorithm. *Pick q such that $r_q > 0$ and let $x_D = \gamma e_q$*, just the unit vector along the qth coordinate axis.

This choice simplifies the optimization even further since x_D is now just proportional the qth column of D. We'll define $u = B^{-1} D e_q$ and $v = B^{-1} b$. Now we have

$$\begin{aligned} \max \quad & \gamma r_D^T e_q \\ \text{s.t.} \quad & \gamma u \leq v \\ & \gamma x_q \geq 0 \end{aligned}$$

Who Exits the Basis - (Udacity)

Of course, the greater the γ, the greater the value, so we want to make this as big as possible. But how big can we make it? Let's make this another quiz.



The answer is the first expression

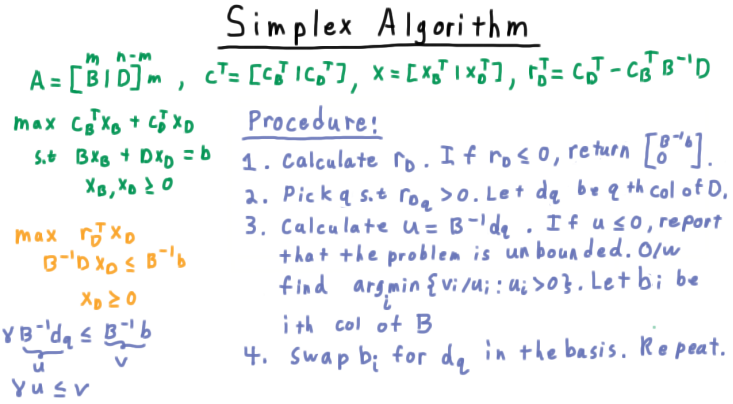
$$\gamma = \min\{v_i/u_i : u_i > 0\}$$

Unless, u_i is positive, we can make γ as big as we want without running into the contraint. Of these contraints, we'll hit the one with this lowest ratio first.

Setting γ to this value makes one of these constraints tight, and sends the corresponding entry of x_B to zero. Remember that this equation came the contraint that x_B be nonnegative. We can then bring d_q into our basis and kick out the column corresponding to the constraint that became tight, and repeat.

Simplex Algorithm - (Udacity, Youtube)

In more detail, we can express the simplex algorithms as follows:



Simplex Example - (Udacity, Youtube)

Simplex Correctness - (Udacity, Youtube)

We have now described the simplex algorithm and seen it illustrated on a simple example.

Next, we argue that the algorithm is correct, giving us a basic feasible solution for bounded linear programs and reporting that unbounded ones are unbounded.

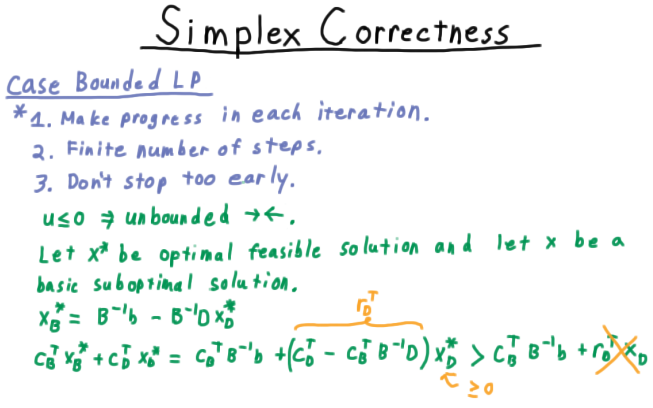
Let's consider the bounded case first. First we recognize that at each step we make some progress, usually improving the objective value. We have to be careful here in the case of degenerate basic solutions. Going back to the algorithm, remember that we pick a new columns to go into the basis because it corresponds to a positive value in r_D, and hence, increasing it, increases the objective value. The trouble is that if the current basic solution is degerate–i.e. v has a zero entry, then it's possible that we won't get to move in this direction at all. The nightmare scenario is that we end up in some kind of cycle.

There are two ways of coping with this challenge. One is to perturb the contraints slightly. The other is to give preference to lower indexed columns both for entering and leaving the basis. This is known as Bland's rule. In either case, we can be assured of making some progress in each step, but the notion is a little tricky.

Clearly, we have a finite number of steps because there are only n choose m possible bases, and because we make progress, we don't cycle among them.

Now, we just need to make sure that we don't stop too early. There are two possible ways the algorithm can terminate: either because u is nonpositive, or because $r_D < 0$. Let's consider the termination because of u first. This turns out to be pretty trivial. If $u \geq 0$, then we get to keep going in the direction of a_q as far as we want, increasing the objective value the whole way, so clearly the problem is unbounded. We won't terminate falsely on that score.

Let's consider the case where we terminate on after examining r_D then. Let x^* be an optimal feasible solution and let x be the current suboptimal basic solution in the simplex algorithm.
Recall that once I choose that basis, I can solve for x_B^* like so,



then substitute back into the objective function. *Note that the choice of basis (partitioning the columns of A into B and D) is done the basis of x, not x^*.*
This expression here is the same r_D that we obtained in the simplex method. Because x is suboptimal, we obtain have a strict inequality when replacing x^* with x.
Note, however, that x_D is zero. Also, x_D^* is nonnegative, so for the inequality to hold at least one entry of r_D must be positive. Hence, if there is a better solution, the simplex algorithm won't terminate.

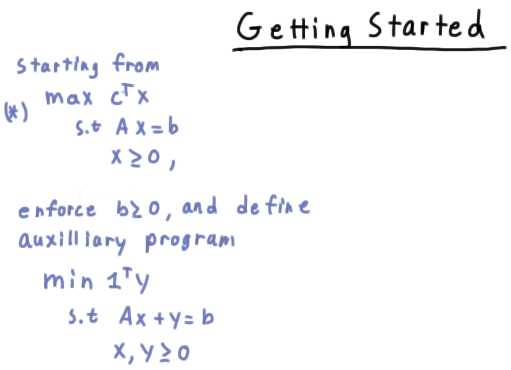
That wraps up the case where the program is bounded. How about when it is unbounded? By the same argument just given, we won't hit the case where $r_D \leq 0$. The algorithm also can't run forever because it avoids cycling, and can thus can only visit each of the n choose m bases once. The only possible remaining outcome is termination after inspecting u, as desired.

Getting Started - (Udacity, Youtube)

If you have been paying careful attention, you will have noticed that the simplex algorithm started with a basic feasible solution. Now in some cases, basic feasible solutions are easy to find just by inspection, but not always.

For these harder cases, we can create an auxilliary program to help us. First we negate the constraint equations as necessary so that we have $b \geq 0$.

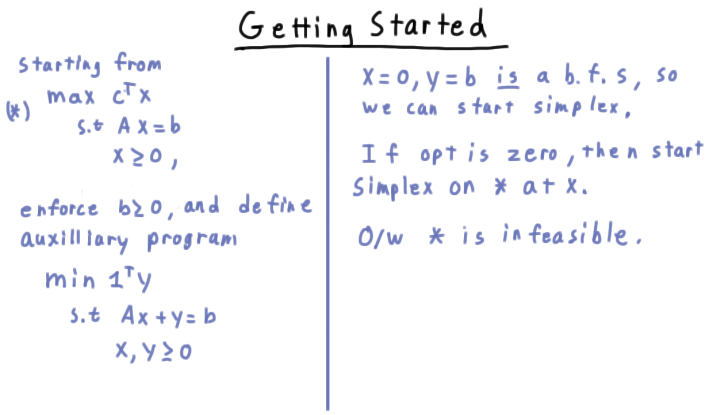
Then we create our auxilliary program as follows.



We introduce artificial variables y that represent the slack between Ax and b, require these variable to be non-negative, and then try to minimize their sum.

For this auxilliary program, it *is* easy to find a basic feasible solution: just set $x = 0$ and $y = b$. Therefore, we can start the simplex algorithm. If we find that the optimum value is zero, then we can start our original program with the values in x. On the other hand, if the

optimum is greater than zero, that means that the original problem was infeasible. This is sometimes called the “Two Phase” approach for solving LPs.



Conclusion - (Udacity, Youtube)

The simplex method as it is described here was first published by George Dantzig in 1947. Fourier apparently had a similar idea in the early 19th century, and Leonid Kantorovich had already used the method to help the Soviet army in World World II. It was Dantzig’s publication, however, that led to the widespread application of the method to industry as the lessons of operations research learned from the war began to be applied to the wider economy and fuel the post-war economic boom. It remains a popular algorithm today.

As practical as the algorithm was, theoretical guarantees on its performance remained poor, and in fact, in 1972 Klee and Minty showed that the worst-case complexity is exponential. It wasn’t until 1979 that a Khachiyan published the ellipsoid algorithm and showed that linear programs can be solved in polynomial time. His results were improved upon in 1984 by Karmarkar, whose method turned out to be practical enough to be competitive with the Simplex method for real-world problems. Both of these algorithms take shortcuts through the middle of the polyhedron instead of always going from vertex-to-vertex.

In the next lecture, we’ll talk about the duality paradigm, which rises out of linear programming and has been the source of many insights and the inspiration for new algorithms. Even with a whole other lesson, however, we are only able to scratch the surface of the huge body of knowledge surrounding this fundamental problem that has shown itself to be of deep importance in both theory and practice.