Master Notes

Professor Marcus

May 26, 2022

The date listed above is the last time it was edited. I will try to post the notes for each lecture before the lecture happens. It may happen that I will want to add to (or change) them based on a question in class.

If that happens, I will highlight the added/changed part like this.

It is also possible that I may not get to all of the material, in which case you should be sure to read it yourself.

If that happens, I will highlight the missed part like this.

It is also possible that there are errors in these notes that I learn about later.

If that happens, I will highlight the corrected part like this.

Note: if you have any issues seeing the difference between these colors, please let me know (I can easily change them).

Contents

1 Week 1: Linear Algebra Review and Intro to Optimization

1.1 Linear Algebra Review

In this class, when we say the word *vector*, we will mean a *column* vector:

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

If I want to talk about an object that looks like

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

then I can write this as $\mathbf{v}^\intercal$ where $\intercal$ is the *transpose* operator (these are sometimes referred to as *covectors*). Note that some sources (like the Bertsimas/Tsitsiklis book) use $\mathbf{A}'$ to denote the transpose of $\mathbf{A}$.

Given a collection of vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\} \subseteq \mathbb{R}^n$, the set of vectors $\mathbf{x} \in \mathbb{R}^n$ that can be written as

$$\mathbf{x} = \sum_i \alpha_i \mathbf{v}_i \tag{1}$$

for some values of $\alpha_i \in \mathbb{R}$ is called the *span* of $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$. Given vectors $\mathbf{v}_i \in \mathbb{R}^n$, the collection of vectors $\mathbf{x} \in \operatorname{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ is called a *linear subspace of* $\mathbb{R}^n$.

In general, a linear subspace can be written as the span of vectors in a lot of different ways. In particular, if $\mathbf{x} \in \operatorname{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$, then

$$\operatorname{span}\{\mathbf{x}, \mathbf{v}_1, \ldots, \mathbf{v}_k\} = \operatorname{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}.$$

This means that some collections of vectors are suboptimal when it comes to describing a linear subspace — in fact, some subset of the vectors would have described the same linear subspace. This "suboptimality" is known as *linear dependence*. That is, we call a collection of vectors V *linearly dependent* if there exists a subset of $U \subset V$ $(U \neq V)$ for which $\operatorname{span}(V) = \operatorname{span}(U)$. If no such subset exists (the collection V is minimal in some sense), then we say that the vectors in V are *linearly independent* and refer to this collection as a *basis*.

One of the fundamental properties of linear subspaces of $\mathbb{R}^n$ is the fact that every basis has the same size. The number of vectors in a basis for a linear subspace is called its *dimension*. In other words,

Theorem 1. *The vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ form a basis if and only if $\operatorname{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ forms a k dimensional space.*

The reason a basis is better than a non-basis is that the description of a vector using a basis is *unique*. As a small example, let

$$\mathbf{x} = \mathbf{u} + 2\mathbf{v} - 3\mathbf{w} \quad \text{and} \quad \mathbf{y} = \mathbf{u} - 7\mathbf{v} + \pi\mathbf{w}$$

In the case that $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ form a basis, we can tell that $\mathbf{x} \neq \mathbf{y}$ just by looking at the coefficients and seeing they are different: $(1, 2, -3) \neq (1, -7, \pi)$. If $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ do not form a basis, then we

don't know whether $\mathbf{x}$ and $\mathbf{y}$ are the same and this is annoying. If I solve a problem and get $\mathbf{x}$ and you solve the same problem and get $\mathbf{y}$, then we might thing one of us is wrong (when in fact we are both right).

One property of linear subspaces that is sometimes useful and sometimes annoying is that they must always contain the vector $\mathbf{0}$. So, for example, Let $S = \{(x, y) : x + y = 1\}$. S is *not* a linear subspace (it does not contain $(0, 0)$, even though the graph of S is line. Furthermore, any line going through $(0, 0)$ is a linear subspace, so the only difference between S and a legitimate linear subspace is that S has been shifted away from $(0, 0)$. Given a linear subspace X and a vector $\mathbf{b} \neq \mathbf{0}$, the collection of vectors

$$S = \{\mathbf{x} + \mathbf{b} : \mathbf{x} \in X\}$$

is called an *affine subspace*. The dimension of S is then said to be the same as the dimension of X (then linear subspace it was formed from).

1.1.1 Operations

First we define matrix–vector multiplication, which can be done in two ways. If we think of a matrix $\mathbf{M}$ as a collection of (column) vectors, then $\mathbf{Mv}$ forms a linear combination of these vectors:

$$\begin{bmatrix} | & | & \dots & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_n \\ | & | & \dots & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \sum_i x_i \mathbf{v}_i.$$

On the other hand, if I think of a matrix as a collection of (rows) covectors, then $M\mathbf{v}$ forms a vector of dot products:

$$\begin{bmatrix} — & \mathbf{u}_1^\intercal & — \\ — & \mathbf{u}_2^\intercal & — \\ \vdots & \vdots & \vdots \\ — & \mathbf{u}_m^\intercal & — \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{u}_1 \cdot \mathbf{x} \\ \mathbf{u}_2 \cdot \mathbf{x} \\ \vdots \\ \mathbf{u}_m \cdot \mathbf{x} \end{bmatrix}$$

In particular, for $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$, we have

$$\mathbf{u}^\intercal \mathbf{v} = \mathbf{u} \cdot \mathbf{v}$$

and this way of writing the dot product will appear *a lot* in this class (so get used to it quickly). This is *not the same* as $\mathbf{u}\mathbf{v}^\intercal$ which would be an $n \times n$ matrix with entries $M_{ij} = \mathbf{u}_i \mathbf{v}_j$ (see (2) below). Fortunately, we will not see much of these in this class, so it should be easier to tell the difference (but this can sometimes cause you to forget that there is a difference).

Finally, we can think of matrix–matrix multiplication as a well-organized collection of matrix–vector multiplications:

$$\mathbf{M} \begin{bmatrix} | & | & \dots & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_n \\ | & | & \dots & | \end{bmatrix} = \begin{bmatrix} | & | & \dots & | \\ \mathbf{M}\mathbf{v}_1 & \mathbf{M}\mathbf{v}_2 & \dots & \mathbf{M}\mathbf{v}_n \\ | & | & \dots & | \end{bmatrix}$$

or via the equivalent formula

$$[\mathbf{AB}]_{i,j} = \sum_k A_{ik} B_{kj}$$

Note that, in particular, for two matrices to be multiplied, the inner dimensions must match (the n in the equation below) and the outer dimensions become the dimensions of the new matrix (the m and the p):

$$\underbrace{\mathbf{A}}_{m\times n}\underbrace{\mathbf{B}}_{n\times p} = \underbrace{\mathbf{AB}}_{m\times p}$$

Or, repeating the examples above:

$$\underbrace{\mathbf{u}^\intercal}_{1\times n}\underbrace{\mathbf{v}}_{n\times 1} = \underbrace{\mathbf{u}\cdot\mathbf{v}}_{1\times 1} \quad \text{and} \quad \underbrace{\mathbf{u}}_{n\times 1}\underbrace{\mathbf{v}^\intercal}_{1\times n} = \underbrace{\mathbf{u}\mathbf{v}^\intercal}_{n\times n} \tag{2}$$

1.1.2 Rank

Given an $m \times n$ matrix $\mathbf{M}$, the collection of vectors

$$\{\mathbf{Mx} : \mathbf{x} \in \mathbb{R}^n\}$$

is called the *image* of $\mathbf{M}$ and the collection of vectors

$$\{\mathbf{x} \in \mathbb{R}^n : \mathbf{Mx} = \mathbf{0}\}$$

is called the *kernel*. Both are linear subspaces. Using the terminology from the first section, the image of $\mathbf{A}$ is the span of the columns of $\mathbf{M}$.

$$\mathbf{M} = \begin{bmatrix} | & | & \dots & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_n \\ | & | & \dots & | \end{bmatrix} \quad \Rightarrow \quad \text{im}(\mathbf{M}) = \text{span}\{\mathbf{v}_1, \dots, \mathbf{v}_n\}.$$

The dimension of the image of a matrix M is called its *rank*. One of the "magical" theorems of linear algebra is that every matrix (no matter what its dimensions) has the same rank as its transpose.

Theorem 2. *For all $n \times m$ matrices $\mathbf{A}$,*

$$\text{rank}(\mathbf{A}) = \text{rank}(\mathbf{A}^\intercal)$$

In particular, this means that, for an $m \times n$ matrix $\mathbf{A}$, $\text{rank}(\mathbf{A}) \leq \min\{m, n\}$. When $\text{rank}(\mathbf{A}) = \min\{m, n\}$, the matrix $\mathbf{A}$ is said to have *full rank*. Or, more specifically, it is said to have *full row rank* when $\text{rank}(\mathbf{A}) = m$ and *full column rank* when $\text{rank}(\mathbf{A}) = n$. One thing to remember about rank is that it is easy to lose it but impossible to get it back (once it is gone). In particular, if $\mathbf{A}$ and $\mathbf{B}$ are matrices for which $\mathbf{AB}$ makes sense, we have the inequality

$$\text{rank}(\mathbf{AB}) \leq \min\{\text{rank}(\mathbf{A}), \text{rank}(\mathbf{B})\}$$

We will denote the $k \times k$ identity matrix as $\mathbf{I}_k$ (or simply $\mathbf{I}$ if it is clear what the dimensions are). A matrix $\mathbf{A}$ is called *invertible* if there exists a matrix $\mathbf{B}$ for which

$$\mathbf{AB} = \mathbf{I} = \mathbf{BA} \tag{3}$$

When such a $\mathbf{B}$ exists, we call it the *inverse* of $\mathbf{A}$ and use the notation $\mathbf{A}^{-1}$ to refer to it. Note that *both* equalities in (3) must hold for $\mathbf{B}$ to be a true inverse. It is possible to have

$$\mathbf{AB} = \mathbf{I} \quad \text{and} \quad \mathbf{BA} \neq \mathbf{I} \tag{4}$$

(for example) in which case $\mathbf{B}$ is not the inverse[1] of $\mathbf{A}$. One of the most important theorems of linear algebra is that one can tell whether a matrix is invertible simply by comparing its rank to its dimensions.

Theorem 3. *An $m \times n$ matrix $\mathbf{A}$ is invertible if and only if $m = n = \text{rank}(\mathbf{A})$.*

In the case that $\mathbf{A}$ is invertible, we can solve systems of linear equations using the inverse:

$$\mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b} \quad \Rightarrow \quad \mathbf{Ix} = \mathbf{A}^{-1}\mathbf{b} \quad \Rightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

In general, the set of solutions to a linear equation

$$\{\mathbf{x} \in \mathbb{R}^n : \mathbf{Ax} = \mathbf{b}\}$$

forms an affine subspace of $\mathbb{R}^n$ (a linear subspace if $\mathbf{b} = \mathbf{0}$) that has dimension $n - \text{rank}(\mathbf{A})$. The case when $\mathbf{A}$ is invertible means $\text{rank}(\mathbf{A}) = n$, so the space of solutions has dimension 0 (that is, it is a single point).

One of the important things to recall about inverses is that the product of two square matrices $\mathbf{A}$ and $\mathbf{B}$ is invertible if and only if $\mathbf{A}$ and $\mathbf{B}$ are both invertible. In this case, we have the formula

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

which follows directly from the definition of inverse.

1.1.3 Hyperplanes

An affine subspace $H \subseteq \mathbb{R}^n$ with $\dim(H) = k$ is said to have *codimension* $n - k$. Of particular interest in this course will be affine subspaces of dimension $n - 1$ (codimension 1), which we will refer to as *hyperplanes*. These are formed (as discussed above) by taking a linear subspace of dimension $n - 1$ (codimension 1) and shifting by a vector. As you perhaps recall, a linear subspace of codimension 1 can be described using its *normal vector*. That is, for a given vector $\mathbf{v} \in \mathbb{R}^n$ with $\mathbf{v} \neq \mathbf{0}$, the set

$$H = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{x} \cdot \mathbf{v} = 0\}$$

is a linear subspace of codimension 1 and all linear subspaces of codimension 1 living inside of $\mathbb{R}^n$ can be written in this way.

A hyperplane is then a shift of one of these linear subspaces. That is, for two given vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ with $\mathbf{u}, \mathbf{v} \neq \mathbf{0}$, a hyperplane in $\mathbb{R}^n$ can be written in the form

$$H' = H + \mathbf{u} = \{\mathbf{x} + \mathbf{u} \in \mathbb{R}^n : \mathbf{x} \cdot \mathbf{v} = 0\}.$$

However, this is not the form you will usually see them. Instead, someone will substitute $\mathbf{y} = \mathbf{x} + \mathbf{u}$ (recall $\mathbf{u}$ is fixed) to get

$$\begin{aligned} H' &= \{\mathbf{x} + \mathbf{u} \in \mathbb{R}^n : \mathbf{x} \cdot \mathbf{v} = 0\} \\ &= \{\mathbf{y} \in \mathbb{R}^n : (\mathbf{y} - \mathbf{u}) \cdot \mathbf{v} = 0\} \\ &= \{\mathbf{y} \in \mathbb{R}^n : \mathbf{y} \cdot \mathbf{v} = t\} \end{aligned}$$

where $t = \mathbf{u} \cdot \mathbf{v}$ is a fixed number (since $\mathbf{u}$ and $\mathbf{v}$ are fixed vectors).

That is, given a fixed vector $\mathbf{v}$ and fixed real number t, the set

$$H' = \{\mathbf{y} \in \mathbb{R}^n : \mathbf{y} \cdot \mathbf{v} = t\}$$

forms a hyperplane in $\mathbb{R}^n$ and all hyperplanes in $\mathbb{R}^n$ can be written in this way. Geometrically, the $\mathbf{v}$ will again be the normal vector to H', with t now telling you how far H' has been shifted in the direction of $\mathbf{v}$.

[1] A matrix $\mathbf{B}$ that satisfies (4) is sometimes called the *right-inverse* or *pseudo-inverse* but we will not use these terms in this class.

1.1.4 A thought on transposes

Someone at the end of class asked a question that made me want to say something about transposes. If the transpose seems like a strange operation that really doesn't fit into normal linear algebra, it is because transposes *are* a strange operation that really doesn't fit into normal linear algebra. We will see later on in this class what the transpose operation represents, but (at least for now) you should not be concerned if the transpose does not make sense to you geometrically like everything else does.

1.1.5 References

1. Bertsimas and Tsitsiklis: Section 1.5
2. Your linear algebra class from your first semester at EPFL![2]

1.2 Intro to Optimization

At its heart, every optimization problem looks like the following:

$$\max\{f(x) : x \in X\} \tag{5}$$

where f is some function (called the *objective function*) and X is some set of things (called the *feasible region*) you are allowed to plug into f. The points in X are called *feasible points* and the goal is to find a feasible point $x_0 \in X$ for which

$$f(x_0) \geq f(x) \text{ for all } x \in X.$$

Some comments:

- Points that are not in X are called *infeasible points*.
- A "largest value" may not exist (this can happen for multiple reasons).
- In the form that it is written, the only possible way to solve this would be to plug in every possible value of $x \in X$ and see which one gives the biggest $f(x)$. In particular, if X is infinite, there is essentially no way to solve this problem (in general).

Fortunately, we are not in the business of trying to optimize "every possible function" over "every possible set X." Typically, we know something about f and X that can make optimization easier (in particular, not impossible). For example, we are sometimes told that f is *continuous* or *differentiable*, in which case we can perform a "hill-climbing algorithm" to try to find the optimal point (at a given point $\mathbf{x}$, take a step in the direction that causes the objective to go up as much as possible). Similarly, we might have a nice description of X like

$$X_1 = \left\{\mathbf{y} \in \mathbb{R}^3 : \|\mathbf{y}\| \leq 1\right\} \quad \text{or} \quad X_2 = \left\{\mathbf{y} \in \mathbb{Z}^3 : \|\mathbf{y}\| \leq 20\right\}.$$

If the feasible region is defined by a set of inequalities, those inequalities are called *constraints*. Despite the only difference between X_1 and X_2 being a change from $\mathbb{R}$ to $\mathbb{Z}$, the two domains are actually quite different, and optimizing over X_2 will be much harder than optimizing over X_1 (for reasons we will see).

It is here that I should mention that there is some strange terminology that is used when it comes to optimization. While we typically think of "solution" as being the answer to a problem,

[2] The second semester (while super useful and interesting for other reasons) is less relevant to this class.

the word *solution* is used in this field to mean "anything that you can to try to plug into your objective function (whether it is feasible or not). Hence we will talk about *feasible solutions* to be those that satisfy the constraints and *infeasible solutions* to be those that do not. Since we are no longer using the word "solution" to talk about the answer to a problem, then we need a new word for this. In fact, depending on the optimization problem, the answer can take one of three forms:

1. There exists an *optimal solution*, which is a feasible solution $\mathbf{x}_*$ that satisfies $f(\mathbf{x}_*) \geq f(\mathbf{x})$ for all $\mathbf{x} \in S$.
2. The problem is *unbounded* — that is there exist feasible solutions for which $f(\mathbf{x})$ can be arbitrarily large.
3. The problem is *infeasible*, which happens when S is an empty set (there are no feasible solutions).

1.2.1 Convex Spaces

There are essentially two types of spaces when it comes to optimizing: convex and non-convex.[3]

A space X is called *convex* if you can get from point $A \in X$ to a point $B \in X$ by following a straight line that never leaves X. Mathematically, X is convex if for all pairs of points $\mathbf{x}, \mathbf{y} \in X$ and all $\lambda \in [0, 1]$ the point

$$(1 - \lambda)\mathbf{x} + \lambda\mathbf{y} \in X$$

Notice that when $\lambda = 0$, we are at $\mathbf{x}$ and when $\lambda = 1$, we are at $\mathbf{y}$, and for λ in between we get a line connecting them. So this is *exactly* how we described it. The reason convex spaces are nice is that the boundaries (if there are any) actually mean something.

For example, consider the space X_1 above. The point $\mathbf{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ is a feasible point, but it is also on the boundary (so it is next to some infeasible points). In particular, if you were to try to move in the direction $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ a tiny bit, so

$$\mathbf{y}' = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 0.001 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

you would notice $\mathbf{y}'$ is infeasible. From this, you can conclude that

$$\mathbf{y}_t = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

is infeasible for all $t \geq 0.001$. That is, if you are walking on a line through a convex space, and you cross the boundary into infeasible territory, you will *never* get back into feasible territory

[3] Note that we are using the word "convex" in a different meaning then when someone talks about a function f being "convex"). The "convexity" we will be considering here is a (geometric) property of sets (not functions). However, there is a link between the two: if a function f is convex (in the function sense) then the sets $\{\mathbf{x} : f(\mathbf{x}) \geq c\}$ are each convex (in the geometric sense) for every $c \in \mathbb{R}$.

again. However this is not the case in the space X_2 from above.

$$\mathbf{y}' = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 0.001 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

is not in X_2, but

$$\mathbf{y}'' = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 17 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

is in X_2.

1.2.2 Hill climbing intuition (and Lagrange multipliers)

In this class, a number of the most useful things we will discover will come from thinking about optimization geometrically. For example, consider the hill-climbing algorithm that was mentioned above. I like to pretend that I am just floating in the space X (and I am happy to just float). There is an objective function which assigns a value to each point in the space, but I don't see those values (or care). HOWEVER, there is an external "being" that sees those values and wants to push you in a direction that will increase the objective function value (like your parents, when they keep suggesting how nice it would be to have a doctor in the family). If f is the objective function (let's assume it is differentiable), then at a given point $\mathbf{x}$, the amount of force that f will apply is equal to $\nabla f(\mathbf{x})$. A force will be applied everywhere in the space (though it may be a different size or direction at different points) and typically this force will cause me to move (and as a result, the objective function will increase). However, if I am at a local maximum (that is, there are no feasible points around me where the objective function is higher than it is at my current spot), then I won't move (I can't move).

Physics has a very nice theory about the relationship between forces and movement — an object moves in the direction of the sum of the forces applied to it. In particular, there is only one way for an object to be at rest — the sum of the forces being applied to it have to equal $\mathbf{0}$. So that must be the case here (assuming I am not moving). It is possible that $\nabla f(\mathbf{x}) = \mathbf{0}$ all by itself (in which case, this would be a normal critical point). The only other possibility is that there are other forces acting on the object that are pushing back against the force that ∇f is applying. Those forces can only come from one place: the boundary of the space. As an example, consider the fact that you are currently under a constant force (gravity) and yet you are not falling towards the center of the earth. The reason gravity isn't moving you when you stand up is that floor is asserting a force on your feet that is equal strength (but opposite direction) to the gravity is. Depending on how you are standing, that "antiforce" could be pushing equally on both feet or it could be pushing different amounts.

So then you might ask, "What force could the boundary possibly be putting on me?" As it turns out, we can figure out the direction of such a force — a piece of boundary defined by the equation $g_i(\mathbf{x}) = 0$ applies its force at a point $\mathbf{x}$ in the direction of $\nabla g_i(\mathbf{x})$ — but we don't necessarily know the amount (so we give it a variable λ_i). In order for these boundary forces to cancel out the force the objective is putting on me, it must be the case that $\nabla f(\mathbf{x}) = \sum_{i=1}^{n} \lambda_i \nabla g_i(\mathbf{x})$, which should look familiar: it is what you likely learned as "the method of Lagrange multipliers". The theory of Lagrange multipliers says

- If I am at a point $\mathbf{x}$ and the force $\nabla f(\mathbf{x})$ is able to push me to a new point, then that new point will have a higher value of f (the gradient of a function points in the direction of maximum increase)

- Therefore, the contrapositive is true — if I am at a point $\mathbf{x}$ which maximizes f (over the set X), it must mean that (even though $\nabla f(\mathbf{x})$ is being applied to me) I am not moving.
- If I am not moving, it must be that the sum of the forces on me in $\mathbf{0}$. That leaves two possibilities: either $\nabla f(\mathbf{x}) = \mathbf{0}$ (meaning this is a normal critical point), or something (the boundaries) are pushing back with forces that cause the sum of forces to be $\mathbf{0}$.
- I know the direction that boundaries can push: $\nabla g_i(\mathbf{x})$, but I don't know the amount. So the best I can say is that the total force exerted by the boundary has the form $\lambda_i \nabla g_i(\mathbf{x})$ for some $\lambda_i \geq 0$.
- Therefore (in order to add up to $\mathbf{0}$), it must be that $\nabla f(\mathbf{x}) = \sum_{i=1}^{n} \lambda_i \nabla g_i(\mathbf{x})$ for some $\lambda_i \geq 0$.

However, there is one detail that, on the surface, seems so obvious that many math books leave it out. When it comes to Lagrange multipliers, however, it can be *super* useful.

- A piece of the boundary can only apply a force against me if I am touching it!

That is, if the boundary is defined by an inequality constraint $g_i(\mathbf{x}) \leq 0$, the only time λ_i is allowed to be nonzero is if $g_i(\mathbf{x}) = 0$!

Your math book might have avoided this part of the discussion by separating into cases of "equality constraints" and "inequality constraints" and then said something like "and then you just have to check all of the points in the boundary of the space defined by the inequality constraints." And while this is (technically) correct, it is hiding a LOT of potential complications in it. For example,

- It might not be easy to check all of the points of a boundary.
- It might not be easy to even find the boundary.

However things we *do* know about the boundary can make our lives simpler. Back to the gravity example, let's assume that I have constraints that describe a weird room with four walls ($g_1, \ldots g_4$), a floor (g_5) and a slanted ceiling ($g6$). Because the ceiling is slanted, $g5$ and $g6$ will intersect (at some point), but they will not intersect in the feasible region (there will be walls separating them). So while Lagrange multipliers will tells you to look for points where

$$\nabla f(\mathbf{x}) = \sum_{i=1}^{6} \lambda_i \nabla g_i(\mathbf{x})$$

it also tells you that you can ignore any points where $\lambda_5 \neq 0$ and $\lambda_6 \neq 0$ at the same time (there is no way for the floor and the ceiling to push on you at the same time.

This makes inequality constraints rather complicated — depending on how they intersect geometrically, some combinations may give false Lagrange multiplier solutions (see the problem set).

1.2.3 Inequality constraints are complicated

The main issue with inequality constraints is that their relevance will often depend on the point you referring to. Read the inequality $x \leq 7$ out loud and notice how you say "less than OR equal to." Some values of x satisfy $x \leq 7$ with equality and other values satisfy it without

equality. Given a feasible point $\mathbf{x} \in X$, we say a constraint is *active* (or *tight*) at $\mathbf{x}$ if it holds with equality and *inactive* (or *not tight*) otherwise. For example, consider the constraints

$$(a)\ x + y \leq 12 \quad \text{and} \quad (b)\ x^2 + y^2 \leq 122$$

Then

- $\begin{bmatrix} 3 \\ 3 \end{bmatrix}$ is feasible with (a) and (b) inactive
- $\begin{bmatrix} 4 \\ 8 \end{bmatrix}$ is feasible, with (a) active and (b) inactive
- $\begin{bmatrix} 1 \\ 11 \end{bmatrix}$ is feasible, with (a) and (b) active
- $\begin{bmatrix} 4 \\ 9 \end{bmatrix}$ is infeasible

Note that an equality constraint is always considered to be active (at a feasible point). The reason that it is good to know whether a constraint is active at a given point goes back to the previous section — active constraints are the only ones that can apply a counteracting force. So in this new language, our conclusion in the previous section can be stated: if we are at a point $\mathbf{x}_*$ which maximizes $f(\mathbf{x})$ over some domain defined by the inequalities $g_i(\mathbf{x}) \geq b_i$, then it must be that $\nabla f(\mathbf{x}_*) = \sum_{i=1}^{n} \lambda_i \nabla g_i(\mathbf{x}_*)$ for some $\lambda_i \geq 0$ where the only $\lambda_i \neq 0$ are those for which g_i is active at $\mathbf{x}_*$ (so $g_i(\mathbf{x}_*) = b_i$).

This is why understanding the geometry of the space X is important. In theory if I had 200 inequality constraints, I could have 2^{200} possible combinations of active/inactive constraints. That is a lot of "checking the boundary." However some constraints can never be active together — if my feasible space was the room I am sitting, the constraints would be the floor, the walls, and the ceiling. But the floor does not touch the ceiling, so they can't be active at the same time. In fact, a rectangular room only has 8 corners, 12 edges, and 6 faces (a total of 26 feasible combinations, instead of $2^6 = 64$). Our goal, then, will be to pick a collection of spaces which are somewhat versatile (so we can potentially solve many different types of problems) but which are simple enough that we can understand the geometry (these will turn out to be *polytopes*).

1.2.4 References

1. Mathematica examples

1.3 Complexity

If there is one thing I can convince you of in this course, it is the fact that complexity is a complicated matter. The reason is that there is no standard way of saying "how good an algorithm is" so we have a bunch of different ways to try to characterize the goodness of a given algorithm. To be clear, I am only talking about *deterministic* algorithms — which is the only type of algorithm we will see in this class.[4] In this class, we will primarily use *big-O* notation to describe complexity relationships. Specifically, for two functions $f(n)$ and $g(n)$, we will say

$$f(n) = O(g(n))$$

[4]This is in contrast to *randomized* algorithms where you allow for decisions to be made using the random number generator.

to mean that there exists a constant c and number n_0 such that

$$f(n) \leq cg(n)$$

whenever $n \geq n_0$. In layman's turn, big-O notation is used to compare the "growth rates" of functions with the caveat that n could be quite large before the true behavior kicks in.

1.3.1 Worst case complexity

The "gold standard" in computer science is *worst case complexity*. This is also the most common type of way to talk about algorithms, so typically if someone just says "this algorithm runs in $O(n^{4/3})$ time", you should assume they are talking about worst case complexity. Worst case complexity is exactly how it sounds — if you have an algorithm, then I will run the algorithm on all possible inputs of a given size, and whichever one is the slowest is the worst case complexity. Of course it is impossible to run an algorithm on all possible inputs, so instead complexities are typically found using a proof[5] In other words, it is a guarantee — if you run this algorithm on an input of size n, then I can guarantee the algorithm will finish in fewer than $f(n)$ steps.

And this is why these are the "gold standard" — because they give you a guarantee. That said, the way it is stated (using big-O notation) is not a particularly great guarantee because we don't know the value of c. A guarantee that a given algorithm will finish in at most $2n^2$ steps is much different than a guarantee that it will finish in at most $10^{100}n^2$ steps. In fact, a guarantee of at most $2n^3$ steps is often much better than a guarantee of $10^{100}n^2$ (even though big-O notation makes the second one look better). But this is a whole separate issue — what is important (at the moment) is that when we say such things, we are talking about a worst case scenario.

And worst case complexity is only really relevant if you keep coming across problems that cause the algorithm to do poorly. It is possible for an algorithm to have very bad worst case complexity (exponential in the size of the input), but the only way to get such a running time is to construct a particular example that forces this particular algorithm to make bad decisions at all times. So the fact that your algorithm *could* take a really long time to finish is not really representative of its general usefulness.

1.3.2 Real life complexity

And if we are talking about "real life" scenarios, there is another aspect that worst case complexity fails to capture and that is what I call the "upper bound phenomenon". The "upper bound phenomenon" is the fact that there is always an implicit upper bound on the amount of time an algorithm can run. So a lot of times you never even get to see the "worst case scenario" because you've given up at that point.

Imagine you were a farmer and you had an algorithm that took in all of the weather conditions and calculated the optimal amount of water to give your plants in the morning. Every night, you could plug in the weather forecast before you go to sleep and then (hopefully) do what the algorithm tells you to do the next morning. So you had to choose between an algorithm that always took 2 days to finish and an algorithm that finished in 4 hours 99% of the time and took 20 years the other 1% of the time, which do you think you should choose? Yes, in terms of worst case complexity, the first algorithm is *much* better. The problem is that, for your use case, 2 days and 20 years are both equally useless. So the first algorithm is useless 100% of the

[5] And these proofs often use the idea of an "adversary" (so if you hear someone talking about an adversary, they are likely talking about worst case complexity.

time and the second one is useless 1% of the time. But worst case complexity cannot capture this.

And you would be surprised how quickly things become useless. In some sense an $O(n^8)$ algorithm is a *huge* improvement on an $O(2^n)$ algorithm (one is in `P` and the other is in `NP`), but if the n I am interested in is anything bigger than 100, then it is quite possible that both of these algorithms are equally useless. The point is that having small worst case complexity is good, but that does not mean that having large worst case complexity is bad. It just means that it *could be* bad. And we are going to run across this theme multiple times in this class, so I wanted to plant the seed now.

1.3.3 References

1. Bertsimas and Tsitsiklis: Section 1.6

2 Week 2: Polyhedra and Linear Inequalities

We will now start talking about a specific optimization problem called a *linear program*. This is the situation where the objective function is linear — that is, $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$ for some vector $\mathbf{c}$ — and where the constraints are linear inequalities (so have the form $\mathbf{a} \cdot \mathbf{x} \geq b$ for some vector $\mathbf{a}$ and some real value b. As we saw in the previous section, knowing the geometry of the feasible region is an important factor for solving optimizations problems, so we start by trying to understand the types of sets that can be formed using these linear inequality constraints.

2.1 Geometry of linear programs (polyhedra)

Geometrically, a linear inequality corresponds to what is known as a *half space* — that is, it consists of an affine hyperplane and all of the points on one side of that hyperplane. The feasible region for a linear program is a finite intersection of half spaces, which is known as a *polyhedron*. As we discussed previously, a "nice" feasible region is one that is convex, so the first thing to do is to determine whether these polyhedra are convex (or not). On the problem set, we showed that the intersection of convex sets is convex, so in order to show that a polyhedron is convex it suffices to show that a half-space is convex. In class, we showed that this is true:

Theorem 4. *Half-spaces are convex (see the video*[6] *for the proof).*

One thing we did see in the proof was that things depended very heavily on the fact that certain numbers were positive. This is because inequalities (unlike equalities) change if you multiply by a negative number. So this means a lot of things that we are used to from linear algebra will become slightly different in the inequality context. For example, in linear algebra you had *linear combinations*

$$\sum_i a_i \mathbf{v}_i \quad \text{where } a_i \in \mathbb{R}$$

which will become *nonnegative linear combinations*

$$\sum_i a_i \mathbf{v}_i \quad \text{where } a_i \in \mathbb{R}, a_i \geq 0$$

or *convex combinations*

$$\sum_i a_i \mathbf{v}_i \quad \text{where } a_i \in \mathbb{R}, a_i \geq 0, \sum_i a_i = 1$$

This leads to different versions of *span*:

- all linear combinations $\Longrightarrow$ the span of the vectors
- all nonnegative linear combinations $\Longrightarrow$ the *cone* of the vectors
- all convex combinations $\Longrightarrow$ the *convex hull* of the vectors

The convex hull of a finite number of points is a *polytope* (a polytope is a polyhedron that is also bounded). And this brings up an interesting point — if I have a polytope that I want to talk about, there are two ways I can describe that polytope:

- as the convex hull of some collection of points

[6] Class 2, Part 1, 26:00

- as the intersection of half-spaces

The interesting thing is that the two descriptions are really *very* different. By that, I mean that if I am doing something that requires me to know the half-space description of a polytope and I only have the convex hull description, I could be in trouble. In general, getting one description from the other is an NP-hard problem, so there may be some approaches to solving linear programs that make sense geometrically, but just do not translate into math well because we don't have the right information available.

2.1.1 References

1. Bertsimas and Tsitsiklis: Section 2.1

2.2 Algebra of linear programs (linear inequalities)

The biggest difference between linear algebra and linear programming is the replacement of equalities with inequalities. For example, you used to solve things like

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{6}$$

whereas now we will try to solve things like

$$\mathbf{A}\mathbf{x} \geq \mathbf{b}. \tag{7}$$

The word "solve" is perhaps not the best word to use... "find all possible solutions to" is more accurate. Recall that in (6), the types of solution sets we would get were pretty simple. Either

1. a solution does not exist, or
2. a unique solution exists, or
3. an infinite set of solutions exists, and these form an affine subspace (which we could describe using basis vectors)

In (7), we will find there are *many* more types of situations that arise. But before that, we should at least define what (7) even means and to do so we will start with (6).

Recall that one way to think about matrix multiplication (the "covector" perspective) was

$$\begin{bmatrix} — & \mathbf{u}_1^\intercal & — \\ — & \mathbf{u}_2^\intercal & — \\ \vdots & \vdots & \vdots \\ — & \mathbf{u}_m^\intercal & — \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{u}_1 \cdot \mathbf{x} \\ \mathbf{u}_2 \cdot \mathbf{x} \\ \vdots \\ \mathbf{u}_m \cdot \mathbf{x} \end{bmatrix}$$

Hence, for example, if

$$\mathbf{A} = \begin{bmatrix} — & \mathbf{u}_1^\intercal & — \\ — & \mathbf{u}_2^\intercal & — \\ — & \mathbf{u}_3^\intercal & — \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

then $\mathbf{A}\mathbf{x} = \mathbf{b}$ corresponds to the system of linear equations

$$\begin{aligned} \mathbf{u}_1 \cdot \mathbf{x} &= b_1 \\ \mathbf{u}_2 \cdot \mathbf{x} &= b_2 \\ \mathbf{u}_3 \cdot \mathbf{x} &= b_3. \end{aligned}$$

Algebraically, we can simply do the same thing:

$$\begin{bmatrix} — & \mathbf{u}_1^\intercal & — \\ — & \mathbf{u}_2^\intercal & — \\ — & \mathbf{u}_3^\intercal & — \end{bmatrix} \mathbf{x} \geq \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

corresponds to the system of linear *inequalities*

$$\begin{aligned} \mathbf{u}_1 \cdot \mathbf{x} &\geq b_1 \\ \mathbf{u}_2 \cdot \mathbf{x} &\geq b_2 \\ \mathbf{u}_3 \cdot \mathbf{x} &\geq b_3. \end{aligned}$$

To see what is happening geometrically, recall that each equation $\mathbf{u}_i \cdot \mathbf{x} = b_i$ forms an affine hyperplane. To "solve for x" then means to find a point (or points) $\mathbf{x}$ which lie in the intersection of all of these hyperplanes. Inequalities like $\mathbf{u}_i \cdot \mathbf{x} \geq b_i$ form what are called *half spaces*. They consist of an affine hyperplane (the border) and then everything on one of the two sides of that hyperplane. So for example, the system

$$\begin{bmatrix} -1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{x} \geq \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$$

forms a triangle in the plane (which was impossible when we just had equalities).

2.2.1 When inequalities get all over the floor

To be clear, the idea to "just replace the $=$ with a $\geq$" idea in the previous section is the mathematical equivalent of telling someone to "just feed the baby" — it is easy to accomplish when you have a really nice, quiet, well-behaved baby, but not all babies are nice, quiet, or well-behaved. Some of them throw their food on the floor and laugh at you. Some of them close their mouth right when you are about to put the spoon in, so the food falls onto their clothes (and then they laugh at you). A "generic baby" can often require some amount of creativity in order to get it to eat — I am told that pretending the spoon is an airplane that needs to come in for a landing is a good technique, but I have no experience in the matter. HOWEVER, I do have some experience in getting a "generic system of linear inequalities" to behave itself and that is what this section is about.

Firstly, what do I mean by a "generic system of linear inequalities"? We will consider 6 different "types" of linear inequalities (here $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{x} = [x_1, \ldots, x_n]^\intercal$ are variables):

(1) $\mathbf{v} \cdot \mathbf{x} \geq b$

(2) $\mathbf{v} \cdot \mathbf{x} \leq b$

(3) $\mathbf{v} \cdot \mathbf{x} = b$

(4) $x_i \geq 0$

(5) $x_i \leq 0$

(6) x_i is free

where a "generic system" can include any combination of linear inequalities of each type.

Note that $x_i = 0$ is not there because that would make life too easy (as soon as we know the value of x_i, we can just plug it in!). The meaning of type (6) is essentially the statement that neither (4) nor (5) need to hold (in the solution to this problem, x_i can be any real number, not just a positive/negative one). In that sense, (6) is more of a "lack of inequality" than an inequality itself, but nonetheless we call it a type. Finally, note also that (4) is a special subcase of (1) and (5) is a special subcase of (2). The reason we differentiate them is that (4) and (5) are in "diagonalized form" (which, as we know from linear algebra is a particularly nice form).

The first thing to notice is that 6 forms is a lot of forms, since presumably our algorithm[7] will have to treat each one in a different way. So our first goal is to show that we really don't need all 6 of these types — that any linear program can be transformed into an equivalent one that is easier to deal with. Optimization problems $\mathcal{P}$ and $\mathcal{P}'$ are said to be *equivalent* if for every feasible solution $\mathbf{x}$ for $\mathcal{P}$, there exists a feasible solution $\mathbf{x}'$ for $\mathcal{P}'$ such that

1. $\mathbf{x}$ and $\mathbf{x}'$ have the same cost (in the corresponding objective functions)
2. $\mathbf{x}$ can be easily constructed from $\mathbf{x}'$ (and the reverse is true as well)

Note that nothing here requires the mapping $\mathbf{x} \mapsto \mathbf{x}'$ to be $1-1$ or even symmetric — it may be that $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{P}$ have the same cost, and that both correspond to $\mathbf{x}' \in \mathcal{P}'$. Similarly, one does not need to be able to construct *all* of the reverse mappings — given $\mathbf{x}'$, one might be able to construct $\mathbf{x}_1$ but not $\mathbf{x}_2$ (that is OK, too). The whole goal here is to ensure that

- $\mathcal{P}$ and $\mathcal{P}'$ have the same optimal value
- Given an optimal solution to $\mathcal{P}$, we can easily construct an optimal solution to $\mathcal{P}'$ (and vice versa).

For reasons we will see later, there are actually two types of "standard forms" that we will aim to achieve.

1. Equality standard form: only type (3) and (4) allowed
2. Inequality standard form: only type (1) allowed

How can we "get rid of" inequalities of a given type? We can't — but we can transform them into other types in a variety of ways:

- turn a (2) into a (1)

  Multiply both sides of the inequality by -1 (this flips the sign). For example,

  $$2x + 3y - 7z \leq 2 \quad \Longrightarrow \quad -2x - 3y + 7z \geq -2$$

- turn a (3) into a (1) and (2)

  Use the fact that $x = y$ is equivalent to $x \leq y$ and $x \geq y$. For example,

  $$2x + 3y - 7z = 2 \quad \Longrightarrow \quad 2x + 3y - 7z \geq 2 \quad \text{and} \quad 2x + 3y - 7z \leq 2$$

  Notice that we now have more constraints than we did before (and that's OK)!

[7] While we haven't even started discussing an algorithm yet, it should always be in the back of our minds as the end product.

- turn a (4) into a (1) or a (5) into a (2)

  Since (4) and (5) are special cases of (1) and (2), we can do this by picking a vector that has a single 1 and the rest 0. So for example,

$$x_2 \geq 0 \quad \Rightarrow \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \cdot \mathbf{x} \geq 0$$

- turn a (6) into two (4)'s

  This is a trick that you should remember, because we will see it again. The idea is to decompose a variable into its positive and negative parts and then force those parts to be positive/negative. For example, if we have the constraints

$$2x + 7y = 4 \quad \text{and} \quad x \geq 0 \quad \text{and} \quad y \text{ free}$$

  then we can decompose y into y_+ and y_- and write $y = y_+ - y_-$ where y_+ and y_- must both be nonnegative. Hence we would get the (new) constraints

$$2x + 7y_+ - 7y_- = 4 \quad \text{and} \quad x \geq 0 \quad \text{and} \quad y_+ \geq 0 \quad \text{and} \quad y_- \geq 0.$$

  Two things to note here: firstly, we now have more (and different) variables than we did before (that's OK as long as the resulting inequalities are equivalent). There is still a small argument one needs to make to show that the corresponding LPs are equivalent (this is left for the reader).

- turn a (1) into a (3) and a (4)

  This is the other trick that you should remember because we will see it again. The idea is to introduce what is called a *slack variable*. You can think of a slack variable as a tool for measuring how close an inequality is to an equality. For example, the inequality $x \leq y$ is satisfied by the two points

$$(x, y) = (1, 2) \quad \text{and} \quad (x, y) = (4, 17,000)$$

  but the first solution is much closer to an equality than the second one (there is less slack).

  The nice thing about slack variables is that there is a super easy way to introduce them, which we show by example. Imagine you start with a type (1) equation

$$2x + 7y \geq 4$$

  and put all of the nonzero stuff on the same side

$$2x + 7y - 4 \geq 0.$$

  This looks like a (4) except for the whole "$2x + 7y - 4$" thing, so introduce a new variable w and set $2x + 7y - 4 = w$. So now you have

$$2x + 7y - 4 = w \quad \text{and} \quad w \geq 0$$

  which, we can then turn into a (3) and a (4) by rearranging:

$$2x + 7y - w = 4 \quad \text{and} \quad w \geq 0.$$

  Note that again we introduce a new variable (that's OK).

2.2.2 Success!

So what have we accomplished?

Theorem 5. *Any linear program with constraints of type (1)–(6) above can be turned into an equivalent linear program with constraints of type (1) only, which can then be written in the form*

$$\mathbf{A}\mathbf{x} \geq \mathbf{b}$$

for some matrix $\mathbf{A}$*, some vector* $\mathbf{b}$ *and some vector of variables* $\mathbf{x}$*.*

Secondly,

Theorem 6. *Any linear program with constraints of type (1) – (6) above can be turned into an equivalent linear program with constraints of type (3) and (4) only, which can then be written in the form*

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

for some matrix $\mathbf{A}$*, some vector* $\mathbf{b}$ *and some vector of variables* $\mathbf{x}$*.*

Keep in mind that (in both cases), each of three of $\mathbf{A}, \mathbf{b}, \mathbf{x}$ that appears in the final representation could be quite different from what was in the original system of inequalities (that's OK, as long as the resulting linear programs are equivalent).

Note that (in theory) we have not made any progress whatsoever — we did a lot of work to take a problem (that we can't solve yet) and to rewrite it in a different form (that we can't solve yet). But (in practice) this is a success because each of these forms will be useful when it comes to the actual solving process. One obvious benefit of the first form (for example) is that it looks exactly like the "intersection of half spaces" geometric perspective that we had earlier. An obvious benefit of the second form, on the other hand, is that it is as close to a linear algebra problem as we could possibly hope to get (all of the "inequality-ness" has been pulled out of the matrix and turned into simple inequalities).

2.2.3 References

1. Bertsimas and Tsitsiklis: Section 1.1

3 Week 3: Vertices, Extreme Points, BFSs

3.1 Vertices, extreme points, and basic feasible solutions (Oh, my!)

We started class by noticing that linear programs had a tendency to end in the "corner" of a polyhedron (at least the ones created by dropping a ball inside a box). This was not *always* true — for example, if the "edge" of the box was parallel to the ground, it was possible to get it to stay in the middle of the "edge", but in that case it was tied with a "corner" when it came to the objective function. So we conjectured:

Conjecture 7. *For every vector* $\mathbf{c}$ *and every polyhedron* P*, there exists a "corner"* $\mathbf{y}$ *of* P *for which* $\mathbf{c} \cdot \mathbf{y} \geq \mathbf{c} \cdot \mathbf{x}$ *for all* $\mathbf{x} \in P$*.*

In particular, this allowed for the possibility that a non-"corner" vector $\mathbf{z}$ could have $\mathbf{c} \cdot \mathbf{y} = \mathbf{c} \cdot \mathbf{z}$ and this is what (we conjectured) is what happens when the ball gets stuck in the middle of the edge (the point it gets stuck at is $\mathbf{z}$).

Of course to prove something like this, we can't just wave our hands and say "you know, a *corner*" — we would need a definition that would mathematically characterize what we intuitively thought a "corner" of a polyhedron should be. In particular, it should be something that allows us to test any point in P and then tells us (definitively) whether that point should be considered a "corner". In fact, it will be useful to have 3 definitions of "corner":

- an "optimization" definition (which we call a *vertex*)
- a "geometric" candidate (which we call an *extremal point*)
- an "algebraic" candidate (which we call a *basic feasible solution (BFS)*)

At this point, you might be concerned that we would need to yell and scream and possibly fight over which definition is the best one for what we are trying to do, but (as it turns out), all three definitions are equivalent (making them all equally good definitions of a "corner"). That is,

Theorem 8. *Let* P *be a polyhedron defined by* $\mathbf{Ax} \geq \mathbf{b}$ *and* $\mathbf{x}$ *a point in* P*. Then*

$$\mathbf{x} \textit{ is a vertex} \iff \mathbf{x} \textit{ is an extremal point} \iff \mathbf{x} \textit{ is a BFS}$$

We will prove 2/3 of this statement below [8] (the final 1/3 — that "$\mathbf{x}$ is a vertex $\Rightarrow$ $\mathbf{x}$ is an extremal point" is part of Problem Set 3). It is worth noting that proving the conjecture would be noticable progress because it would effectively reduce a (potentially) infinite problem to a (potentially) finite one. The reasoning is that, since we are in $\mathbb{R}^n$, if a polytope was defined by m constraints, there would be at most $\binom{m}{n}$ possible BFS's and checking each one of those would take a finite (long, but finite) amount of time. Without this conjecture, we could potentially need to check every point in the polyhedron, something that would take a (much) longer time.

3.1.1 Proof: Equivalence of Vectors, Extreme Points, and Basic Feasible Solutions

Now is when we start to look more like a math class (with definition and theorems and proofs and stuff). First the definitions: given a polyhedron $P \subseteq \mathbb{R}^n$, a point $\mathbf{x} \in P$ is called

1. a *vertex* if there exists a linear program for which $\mathbf{x}$ is a unique solution. That is, there exists a vector $\mathbf{c}$ such that $\mathbf{c} \cdot \mathbf{x} > \mathbf{c} \cdot \mathbf{y}$ for all $\mathbf{y} \neq \mathbf{x} \in P$ (note that is equivalent to find a $\mathbf{w}$ for which $\mathbf{w} \cdot \mathbf{x} < \vec{e} \cdot \mathbf{y}$, since we can make $\mathbf{c} = -\mathbf{w}$.

[8] See also the video from Class 3, Part 1 at 34:20.

2. an *extreme point* if it is not the convex combination of two other points in P. That is, the equation $\mathbf{x} = \lambda\mathbf{y} + (1-\lambda)\mathbf{z}$ has no solution satisfying $\mathbf{y}, \mathbf{x} \neq \mathbf{x}$ and $\lambda \in [0,1]$.

3. a *basic feasible solution (BFS)* if it is the unique point of intersection of a set of constraint hyperplanes. In other words, if there exist n linearly independent constraint vectors that are *active* at $\mathbf{x}$. Note that equality constraints must be active in order for the point to be feasible.

And now some proofs. For those coming from outside of the math department (that may not have as much experience proving things as the math students), I will try to make comments about how one would come up with these sorts of proofs on their own as I go.

Theorem 9. *If* $\mathbf{x}$ *is a BFS, then* $\mathbf{x}$ *is a vertex.*

Proof. Assume that P is defined by $\mathbf{Ax} \geq \mathbf{b}$ and that $\mathbf{x}$ is a BFS[9] This means that there exist n linearly independent rows of A which are active constraints. Without loss of generality (changing the order of the constraints does not change the P), we can assume that it is the first n rows.[10] Hence we have

$$\mathbf{A} = \begin{bmatrix}\mathbf{A}_0 \\ \mathbf{A}_1\end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix}\mathbf{b}_0 \\ \mathbf{b}_1\end{bmatrix}$$

where

1. $\mathbf{A}_0$ is $n \times n$ and full rank

2. $\mathbf{b}_0$ has length n

3. $\mathbf{x}$ satisfies
$$\mathbf{A}_0\mathbf{x} = \mathbf{b}_0{}^{11} \quad \text{and} \quad \mathbf{A}_1\mathbf{x} \geq \mathbf{b}_1{}^{12}$$

Now notice[13] that for any $\mathbf{y} \in P$, we have

$$\mathbf{A}_0\mathbf{y} \geq \mathbf{b}_0 = \mathbf{A}_0\mathbf{x}$$

which means $\mathbf{A}_0(\mathbf{y}-\mathbf{x}) \geq \mathbf{0}$ (it is a bunch of nonnegative numbers). On the other hand, $\mathbf{A}_0$ is invertible[14] so

$$\mathbf{A}_0(\mathbf{y}-\mathbf{x}) = \mathbf{0} \quad \Rightarrow \quad (\mathbf{y}-\mathbf{x}) = \mathbf{0} \quad \Rightarrow \quad \mathbf{y} = \mathbf{x}$$

That is, if $\mathbf{y}$ is in P then $\mathbf{A}_0(\mathbf{y}-\mathbf{x})$ is a nonnegative vector, and the only time it is zero is if $\mathbf{x} = \mathbf{y}$[15]. Now define

$$\mathbf{c}^\intercal = \mathbf{1}^\intercal\mathbf{A}_0 \quad \text{where } \mathbf{1} \text{ is the vector in } \mathbb{R}^n \text{ with a 1 as each entry}$$

then

$$\mathbf{c} \cdot (\mathbf{y}-\mathbf{x}) = \mathbf{1}\mathbf{A}_0(\mathbf{y}-\mathbf{x}).$$

[9] If your hypothesis includes something that has a definition, it is a good guess that the first thing you do is apply that definition.

[10] This is just to make notation easier — in theory these constraints could be anywhere, but since changing the order of the constraints of a polyhedron doesn't change the polyhedron, I can simply rearrange them so (regardless of where they were) they are now on top, which means I can use block matrix notation.

[11] These are the basic constraints we know to be active.

[12] These are all of the rest of the constraints (they may or may not be active, but they must be satisfied since $\mathbf{x}$ is feasible)

[13] The key observation is that $\mathbf{A}_0$ is still a matrix of constraints.

[14] The main characteristic of a BFS is the linear independence, so we should expect that to show up somewhere!

[15] This is starting to sound like the definition of vertex, but we are still going to need to produce a vector $\mathbf{c}$ somehow (and this is where a bit of cleverness is needed)

Since for a vector $\mathbf{v}$, $\vec{1}^\intercal \mathbf{v}$ is simply the sum of the elements of $\mathbf{v}$, we then have that $\mathbf{c} \cdot (\mathbf{y} - \mathbf{x})$ is simply the sum of the entries of $\mathbf{A}_0(\mathbf{y} - \mathbf{x})$. But those entries are nonnegative (and are only $\vec{0}$ when $\mathbf{x} = \mathbf{y}$), so that means

$$\mathbf{c} \cdot (\mathbf{y} - \mathbf{x}) \geq 0 \quad \text{with equality only when } \mathbf{x} = \mathbf{y}$$

which is equivalent to the definition of $\mathbf{x}$ being a vertex (so we are done).

□

Before giving the next proof, I want to point out that, were you told to prove this on your own, your first thought should be "I should try to prove the contrapositive". The reason is that the definition of extremal point is more like an anti-definition. So knowing that a given $\mathbf{x}$ is an extremal point is not super useful when proving things, because it doesn't give you something you can put your hands on (at least not until we prove these equivalences). Knowing that a given $\mathbf{x}$ is *not* an extremal point, however, is more useful because it means that $\mathbf{y}$ and $\mathbf{z}$ and λ all exist, and these are things you can then try to manipulate.

Theorem 10. *If* $\mathbf{x}$ *is an extremal point, then* $\mathbf{x}$ *is a BFS.*

Proof. We prove the contrapositive — that is, if $\mathbf{x}$ is not a BFS, then it is not an extremal point. As before, we can assume (without loss of generality) that the active constraints are in the top rows of A, only this time, we will take all of them. That is,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_0 \\ \mathbf{b}_1 \end{bmatrix}$$

where this time $\mathbf{A}_0$ is *not* full rank (since $\mathbf{x}$ is not a BFS) and $\mathbf{A}_1\mathbf{x} > \mathbf{b}_1$.[16] Since $\mathbf{A}_0$ is not full rank, there exists a vector $\mathbf{d} \neq \vec{0}$ in its kernel.[17] Now form the vectors[18]

$$\mathbf{y} = \mathbf{x} + \epsilon\mathbf{d} \quad \text{and} \quad \mathbf{z} = \mathbf{x} - \epsilon\mathbf{d}$$

It is easy to check that $\mathbf{x} = \frac{1}{2}(\mathbf{y} + \mathbf{z})$ so if we can show that $\mathbf{y}$ and $\mathbf{z}$ are both in P (for some small ϵ), this will show that $\mathbf{x}$ is not an extremal point. [19] So we compute[20]

$$\mathbf{A}\mathbf{y} = \mathbf{A}\mathbf{x} + \epsilon\mathbf{A}\mathbf{d} = \begin{bmatrix} \mathbf{b}_0 \\ \mathbf{A}_1(\mathbf{x} + \epsilon\mathbf{d})] \end{bmatrix}$$

and similarly for $\mathbf{z}$. So to prove $\mathbf{y}, \mathbf{z} \in P$, we need to show that there exists an ϵ for which

$$\mathbf{A}_1(\mathbf{x} \pm \epsilon\mathbf{d}) \geq \mathbf{b}_1$$

or (equivalently)

$$\mathbf{A}_1\mathbf{x} - \mathbf{b}_1 \geq \pm\epsilon\mathbf{A}_1\mathbf{d} \tag{8}$$

[16] Again, these are the remaining constraints, but this time $\mathbf{A}_0$ contains all active constraints so we know none of these are active.

[17] Note that "not full rank" is another "lack of something" property. Fortunately, this time there is a theorem from linear algebra which gives us something we can put our hands on.

[18] To prove that something is not an extremal point, we need to two vectors and $\mathbf{d}$ is the only thing we have available

[19] Note that the best (some might say "only") way to show $\mathbf{y}$ is in P, is to show $A\mathbf{y} \geq \mathbf{b}$ (similarly $\mathbf{z}$).

[20] Keeping in mind all the properties $\mathbf{A}_0$ has: $\mathbf{A}_0\mathbf{x} = \mathbf{b}_0$ and $\mathbf{A}_0\mathbf{d} = \mathbf{0}$ means we can calculate $\mathbf{A}_0\mathbf{y}$.

However, we have seen that $\mathbf{A}_1\mathbf{x} - \mathbf{b}_1 > 0$[21] so if δ is the smallest element of $\mathbf{A}_1\mathbf{x} - \mathbf{b}_1$ and K is the largest (in absolute value) element of $\mathbf{A}_1\mathbf{d}$ then setting $\epsilon = \delta/K$. Gives

$$\pm\epsilon\mathbf{A}_1\mathbf{d} \leq \delta\vec{1} \leq \mathbf{A}_1\mathbf{x} - \mathbf{b}_1$$

which shows (8) and finishes the proof. □

One final comment — when it comes to proving the final 1/3 of this, it may be tempting to look at these proofs and try something similar. Don't do that! The reason these proofs are the way they are is because the theorems we wanted to prove involved BFS's and talking about BFS's requires matrices. The statements "$\mathbf{x}$ is a vertex" and "$\mathbf{x}$ is an extreme point" are both geometric statements, so you should try to find a relationship between those two that doesn't require matrices (and will end up being simpler than either of the proofs here).

3.1.2 References

1. Bertsimas and Tsitsiklis: Section 2.2

3.2 Optimal solutions to LPs can always be found at extreme points

Now that we have a characterization of corners, the goal will be to show that it suffices to look for optimal solutions in these corners:

Theorem 11. *Consider the problem of minimizing $\mathbf{c} \cdot \mathbf{x}$ for $\mathbf{x} \in P$ where P is a polyhedron (intersection of halfspaces). If*

1. *there exists an optimal solution in P, and*
2. *P contains at least one extreme point*

Then there exists an extreme point in P which gives the optimal value.

The proof from the lecture[22] uses an idea called *ray-tracing* that consists of two observations:

1. if $\mathbf{x}$ is not a BFS, it means there is a direction which is orthogonal to the normal vectors of the active constraints. If we move in that direction, the constraints that were active at $\mathbf{x}$ will remain active.
2. If $\mathbf{x} \in P$ and we move along a line $\mathbf{x} + \lambda\vec{d}$, one of two things must happen: either we leave the polyhedron at some point or we don't. The situation where we don't leave P we referred to as *P containing a line* situation where we do leave P at some point $\vec{y}$, then it means we had to cross a new constraint at $\vec{y}$ and that constraint must be linearly independent from the current ones (assuming we picked the direction $\vec{d}$ like we said in the first part).

This led to the theorem:

Theorem 12. *The following are equivalent:*

[21] If $\mathbf{v} > 0$, then it has a smallest element. The idea will be to pick ϵ small enough so that

$$\pm\epsilon\mathbf{A}_1\mathbf{d} \leq \delta\mathbf{1}$$

is even smaller than that smallest element.

[22] See video of Class 3, part 2.

1. *P contains at least one extreme point*

2. *P does not contain a line*

3. *There exist n constraints defining P which are linearly independent*

The equivalence of the first and last was Problem 4 in Problem Set 2, and the fact that they are equivalent to the second one uses a similar idea. The nice thing about "containing a line" is that it was geometrically intuitive. For example, let Q be a sub-polyhedron of P (that is, Q contains the same inequalities as P but also additional ones). Saying that P has an extreme point implies Q has an extreme point is not visually obvious, because everything depends on where the new constraints are. However, saying that P does not contain a line implies Q does not contain a line is much more obvious because it is a statement about subsets of subsets.

And this is the main idea of the proof of Theorem 11. Assume $\mathbf{c} \cdot \mathbf{x}$ achieves a minimum on P, and let that minimum value be t. Since P has an extreme point, it does not contain a line. Hence the sub-polyhedron

$$Q = \{\mathbf{x} \in P : \mathbf{c} \cdot \mathbf{x} = t\}$$

also does not contain a line (and therefore has an extreme point $\mathbf{x}_*$). The remainder of the proof is showing that $\mathbf{x}_*$ is also an extreme point of P.

The proof that $\mathbf{x}_*$ is an extreme point of P will go by contradiction [23]. Assume that $\mathbf{x}_*$ is not an extreme point in P — that is, we assume that there exist $\mathbf{y}, \mathbf{z}$ in P for which

$$\mathbf{x}_* = \lambda \mathbf{y} + (1 - \lambda)\mathbf{z} \tag{9}$$

for some $0 \leq \lambda \leq 1$. The goal will be to show that $\mathbf{y}$ and $\mathbf{z}$ must also be in Q — if we can do that, it would imply that $\mathbf{x}_*$ was *not actually* an extreme point in Q, and that would be a contradiction (because we picked $\mathbf{x}_*$ to be an extreme point in Q in the first place).

If you haven't seen this before, it may look weird at first, but it is something we will use a lot in this class, so make sure to take note. The idea is to show that the only way for certain inequalities to be true is if they are actually equalities. For example, consider the following set of inequalities:

$$a \leq c \quad \text{and} \quad b \leq d \quad \text{and} \quad a + b = c + d$$

The only way for all three to be true[24] is if $a = c$ and $b = d$.

For our purposes, we will use the fact that t is the minimum possible cost for any points in P:

$$\mathbf{c} \cdot \mathbf{y} \geq t \quad \text{and} \quad \mathbf{c} \cdot \mathbf{z} \geq t \tag{10}$$

and, on the other hand, we have by (9)

$$t = \mathbf{c} \cdot \mathbf{x}_* = \lambda(\mathbf{c} \cdot \mathbf{y}) + (1 - \lambda)(\mathbf{c} \cdot \mathbf{z}) \tag{11}$$

Together, (10) and (11) can only be true if

$$\mathbf{c} \cdot \mathbf{y} = t \quad \text{and} \quad \mathbf{c} \cdot \mathbf{z} = t.$$

But that would mean (by definition) that $\mathbf{y}, \mathbf{z} \in Q$, which is a contradiction.

As we noted, this is a success because it turns what seems like could be an infinite problem into a finite problem — all we have to do is check $\mathbf{c} \cdot \mathbf{x}$ at all extremal points and see which one is best. Well, not really — there are still some issues like what happens if there is no finite optimal

[23] See the proof of Theorem 10 if you want to know why this could be a good approach to try.

[24] This could be proved by another "proof by contradiction" argument, if you want to be rigorous.

solution or if there are no extreme points in P. And even if those things were not an issue, it was still not a *great* idea to use this "check all extremal points" algorithm because the only method we have for finding extremal points is to check combinations of n linearly independent constraints, and there are potentially A LOT of these combinations. Furthermore, many of them are infeasible, so it seems like a waste of effort to find them all. Particularly when we just discussed a method for finding basic feasible solutions that never left the polyhedron at all (ray-tracing!).

However, there is still a lot that we can add to this ray-tracing idea to make it better — for example, if we are trying to minimize $\mathbf{c} \cdot \mathbf{x}$ then we could limit our attention to those BFSs that have a smaller solution that the one we are currently at, which could cut out a lot of possible BFSs that would clearly be worse. So this will be the plan for next class — to formalize this idea of "ray-tracing in a good direction" into an algorithm.

3.2.1 References

1. Bertsimas and Tsitsiklis: Section 2.5, 2.6

4 Week 4: Simplex!

We start by separating the "Solve a linear program" problem into 2 steps:

1. Find a point in the feasible region
2. Find an optimal solution

We will ignore the first step (at least for now). The reason is that we already have an idea for how to find an optimal solution if we knew a point in the feasible region: Firstly, we could ray-trace to a BFS. Then we could ray-trace from BFS to BFS in directions that improved the objective function. The reason we might hope that we can move from BFS to BFS efficiently is that these correspond to edges in the polyhedron. So (at least geometrically) it is easy to imagine how this might go. Of course there are all kinds of "edge cases" (no pun intended) that we will need to deal with, but this will be the main idea of our algorithm, which is known as the *simplex algorithm*.

4.1 The basic idea

Since we understand the geometry of the inequality standard form, the obvious thing would be to try to accomplish this "moving along edges" procedure there. Assume we have a polyhedron $P \subseteq \mathbb{R}^n$ defined by the constraints

$$\mathbf{A}\mathbf{x} \geq \mathbf{b}$$

where $\mathbf{A}$ is a tall skinny matrix (say $m \times n$ where $m > n$) and assume we are able to find a BFS $\mathbf{y}$. Then we could find the constraints (rows of $\mathbf{A}$) that are active at $\mathbf{y}$ — we call these rows a *basis* because they form a basis for $\mathbb{R}^n$[25] An edge is defined by $n-1$ active constraints, so to move along an edge, we will need to make one of our active constraints inactive (and thus remove that constraint from the basis). We then travel along that edge until we run into a new constraint, which when added to the basis gives us a new BFS. And then we iterate (not forever, of course — we'll discuss the stopping criteria in a moment).

So now let us see how this translates into equality standard form (which will have its own advantages, as we will see). A general equality standard form problem has a polyhedron defined by the constraints

$$\begin{aligned} \mathbf{U}\mathbf{y} &= \mathbf{v} \\ \mathbf{y} &\geq \mathbf{0} \end{aligned}$$

where $\mathbf{U}$ is a short and fat matrix (say $M \times N$ with $N > M$). Note that I am using different values from above $\mathbf{U}$ vs. $\mathbf{A}$ (for example) to make it clear that if we had turned our inequality standard form into an equality standard form, the matrices would be different. Specifically,

$$\mathbf{U} = \begin{bmatrix} \mathbf{A} & -\mathbf{I}_{m\times m} \end{bmatrix} \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} \mathbf{x} \\ \mathbf{w} \end{bmatrix}. \tag{12}$$

HOWEVER, we are going to consider the case that we came upon an LP in equality standard form that we don't know where it came from; in particular, we will assume that $\mathbf{U}$ is just some arbitrary $M \times N$ matrix with $N > M$ (not necessarily in the form of (12)). Since $\mathbf{U}$ has N columns, the associated linear program will have N variables, so a basic solution will need N linearly independent active constraints. But unlike before, some of these constraints are going

[25] It is possible that there are more than n active constraints, but we will ignore that possibility for the moment and come back to it later.

to always stay active (since we have equalities). In fact M of the constraints will remain active, which means we will need to find $N - M$ more active constraints in order to construct a basic solution. By process of elimination, those $N - M$ constraints must be nonnegativity constraints (which is what we call the $\mathbf{y} \geq \mathbf{0}$ constraints). Note that saying a nonnegativity constraint $y_i \geq 0$ is "active" is the same as saying $y_i = 0$, so this would amount to locking down $N - M$ of the variables at 0.

Of course we still need all of the active constraints to be linearly independent! Fortunately, some of the linear independence is obvious:

1. the nonnegativity constraints are linearly independent from each other because they come from a big identity matrix

2. the equality constraints are linearly independent from each other because the slack variables form an identity matrix

So we just need to check that the equality constraints are linearly independent from the nonnegativity constraints. For this, we can do Gaussian elimination — for each nonnegativity constraint, we can "zero-out" the column that has a single 1 in it — and what we are left with is the following lemma:

Lemma 13. *Let $S \subset \{1, \ldots, N\}$ with $|S| = N - M$. The set of constraints*

$$\{y_i = 0 : i \in S\}$$

is linearly independent from the equality constraints if and only if the $M \times M$ submatrix of $\mathbf{U}$ formed by removing the columns in S is invertible.

So for reasons I still cannot explain (and has led to many years of confusion for me) we defined a *basic column* to be a column that is indexed by a variable which is *not* an active constraint.[26] So, to be clear:

- When we are in inequality standard form, the "basis" is a "row basis" which consists of the rows that contain active constraints.

- When we are in equality standard form, the "basis" is a "column basis" which consists of columns associated to the variables that are *inactive* (that is, allowed to be nonzero).

I could spend the rest of this section apologizing for the absurdity of these two statements, but instead I will note that if we just think about the dimensions of things, this actually does make sense. The whole point of requiring linear independence is so that we can get an invertible matrix. Invertible matrices are square. So if $\mathbf{A}$ is an $m \times n$ matrix with $m > n$, an invertible submatrix is going to be formed from n linearly independent rows. On the other hand, $\mathbf{U}$ is an $M \times N$ matrix with $N > M$, so an invertible submatrix in this case is going to be formed from M linearly independent columns. To find those columns, we use the fact that a total of N linearly independent active constraints are needed, with M of them being equality constraints. This leaves $N - M$ nonnegativity constraints that must be active (that is $N - M$ variables that are locked down at 0. Since there are N columns total in $\mathbf{U}$, it makes sense that those M columns are the ones that are *not* associated to the $N - M$ rows.

[26] Note that the book refers to the columns by their variables, so column u_k corresponds to variable x_k. Furthermore, it often will take the analogy further and say the basis contains *the variables*, but that really means the columns corresponding to those variables.

In any case, we still have some translating to do and from what we just saw, we should expect much of it to be "flipped" in some way. In particular, the way to travel along an edge in the row basis was to remove a constraint from the basis. The way to travel along an edge in the column basis will be to *add* a column to the basis. And while this may not be as intuitive as the row basis case, the fact that we know that what we are doing (geometrically) is moving along an edge turns out to be all we need in order to do our calculations (at least if we are clever about it). Geometrically, moving along an edge means we need to move from $\mathbf{y} \to \mathbf{y} + \theta\mathbf{d}$ for some vector $\mathbf{d}$ (which we will now have to figure out). Furthermore, $\mathbf{d}$ should (in some way) correspond to a column u_k entering the basis.

So let's simply list the things we know:

- At the start, u_k is not in the column basis, so $x_k = 0$. In order to get u_k into the basis, we need to make it[27] nonzero, so we need d_k to be nonzero. Since we are free to scale our direction vector however we want, we can just assume that $d_k = 1$.
- All other columns u_i that are not in the basis should stay out of the basis, so for these we want to keep $x_i = 0$ — the only way to do this to have $d_i = 0$ for each nonbasic column u_i where $i \neq k$.
- Lastly, we know that as long as we are traveling along an edge, we are going to remain feasible, which means that
$$\mathbf{U}(\mathbf{y} + \theta\mathbf{d}) = \mathbf{b}$$
for some $\theta > 0$. In particular, this tells us that we need to have $\mathbf{Ud} = \mathbf{0}$.

Now I claim $\mathbf{d}$ is completely defined. In the first two steps, we assigned values to $n - m$ of the entries of $\mathbf{d}$ (either 0 or 1). In the third step, we saw that $\mathbf{d}$ must satisfy $\mathbf{Ud} = \mathbf{0}$, which is a collection of m constraints. Filling in the values from the first two steps, this leaves m (linearly independent, since these are basis columns) constraints in m variables, which means there is a unique solution. So, in some sense, the equation $\mathbf{Ud} = \mathbf{0}$ has dictated what the other entries of $\mathbf{d}$ have to be.

4.2 An example

For example, take the 3×5 matrix $\mathbf{U}$ with columns

$$\mathbf{U} = \begin{bmatrix} | & | & | & | & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3 & \mathbf{u}_4 & \mathbf{u}_5 \\ | & | & | & | & | \end{bmatrix}$$

and let's imagine that our current basis is $\beta = \{\mathbf{u}_2, \mathbf{u}_3, \mathbf{u}_5\}$. The submatrix corresponding to these columns (called the *basis matrix*) is therefore

$$\mathbf{B} = \text{col}_\beta(\mathbf{U}) = \begin{bmatrix} | & | & | \\ \mathbf{u}_2 & \mathbf{u}_3 & \mathbf{u}_5 \\ | & | & | \end{bmatrix}$$

and this is invertible (since a basis contains linearly independent vectors). The BFS $\mathbf{y}$ that corresponds to β would have all of the nonbasis entries 0 and the other entries determined by

[27] At least, we need to allow it the opportunity — it may stay at 0 for reasons related to degeneracy.

the basis matrix:

$$\mathbf{y} = \begin{bmatrix} 0 \\ y_2 \\ y_3 \\ 0 \\ y_5 \end{bmatrix} \quad \text{where} \quad \begin{bmatrix} y_2 \\ y_3 \\ y_5 \end{bmatrix} = \mathbf{B}^{-1}\mathbf{v}.$$

The direction vector $\mathbf{d}$ that corresponds to adding $\mathbf{u}_1$ to the column basis would be

$$\mathbf{d} = \begin{bmatrix} 1 \\ d_2 \\ d_3 \\ 0 \\ d_5 \end{bmatrix} \quad \text{where} \quad \mathbf{d}_\beta = \begin{bmatrix} d_2 \\ d_3 \\ d_5 \end{bmatrix} = -\mathbf{B}^{-1}\mathbf{u}_1$$

If instead we wanted to add $\mathbf{u}_4$ into the basis, we would form the vector

$$\mathbf{d} = \begin{bmatrix} 0 \\ d_2 \\ d_3 \\ 1 \\ d_5 \end{bmatrix} \quad \text{where (this time)} \quad \mathbf{d}_\beta = \begin{bmatrix} d_2 \\ d_3 \\ d_5 \end{bmatrix} = -\mathbf{B}^{-1}\mathbf{u}_4$$

4.2.1 Picking the column to enter the basis

Now that we know which directions we can travel in, the obvious question is which direction to pick*. Note that my use of the term pick* is not a typo — the reason for this is that "picking things" is not a trivial task[28]. And while different picking routines can cause different global behaviors, it doesn't make sense to worry about global behavior now (when we are still figuring out the nuts and bolts of how simplex works). And at the same time, this whole "moving down an edge" thing that simplex is going to do will be (theoretically) the same regardless of which direction we do pick. So (at least for now) pick* will be a black box picking algorithm that we can worry about later.

However, there is one obvious thing that we should demand from pick* — it should pick something that improves the objective function! So given our basis β, we take each $u_k \notin \beta$ and form a direction vector $\mathbf{d}$ (which we saw above is)

- $d_k = 1$ (this forces x_k into the (new) basis)
- $d_i = 0$ for all other non basic variables (this keeps these other x_i out of the (new) basis)
- $\mathbf{d}_\beta = -\mathbf{B}^{-1}\,\mathrm{col}_k(\mathbf{U})$ (this keeps us feasible)

where $\mathbf{B} = \mathrm{col}_\beta(\mathbf{U})$. So now we should figure out if moving in that direction is going to help. To do this, we will calculate what is called the *reduced cost* for the basis β when adding the column u_k (which is nothing more than the derivative of the cost function in the direction $\mathbf{d}$):

$$\bar{c}_k = \mathbf{c} \cdot \mathbf{d} = \mathbf{c}_\beta \cdot \mathbf{d}_\beta + c_k = -\mathbf{c}_\beta^\intercal \mathbf{B}^{-1}\,\mathrm{col}_k(\mathbf{U}) + c_k.$$

So if our goal was to maximize $\mathbf{c} \cdot \mathbf{x}$, we would want to pick a column u_k (variable y_k) which has $\bar{c}_k > 0$ and if our goal was to minimize $\mathbf{c} \cdot \mathbf{x}$, we would want to pick a column/variable which has $\bar{c}_k < 0$. So for now we will assume that pick* gives us such a column/variable (assuming it exists).

[28] This is why we need things like the Axiom of Choice.

4.2.2 Picking the column to leave the basis

For the moment, let's assume such a u_k exists (one that improves the objective function) and let $\mathbf{d}$ be the vector pointing in that direction. We still need to figure out how far to move (the value of θ in $\mathbf{y} + \theta\mathbf{d}$). One thing that is nice about $\mathbf{d}$ is (by the way we constructed it), $\mathbf{U}(\mathbf{y} + \theta\mathbf{d}) = \mathbf{b}$ for all θ, so the only constraints we need to worry about are the nonnegativity constraints $y_i \geq 0$. We claim there are two possibilities:

1. all elements of $\mathbf{d}$ are nonnegative — then adding $\theta\mathbf{d}$ is just going to make each y_i bigger, so we will never get a new $y_i = 0$. Then we can let $\theta \to \infty$ (knowing we are still feasible), which will drive the objective function as high (or low) as we want. In other words, this linear program has no optimal solution.

2. there exists at least one j with $d_j < 0$ — then $y_j + \theta d_j$ will eventually reach 0, and we can easily solve for when that happens:

$$\theta_j = \frac{-y_j}{d_j}.$$

   For θ smaller than this, the new value of y_j will be feasible and for θ larger than this, the new value will be infeasible. So in order to keep everything feasible, we need to pick*[29] the smallest such θ_j:

$$\theta_* = \min_{i:d_i<0} \frac{-y_i}{d_i}$$

   and this will ensure that all variables remain feasible in $\mathbf{x} + \theta_*\mathbf{d}$. Furthermore, whichever index resulted in the smallest value of θ (let's say it was y_t), will now have $y_t = 0$, which means we can take it out of the basis.

So, what have we accomplished? Well, it is possible that we found a direction which proved the linear program is unbounded, which would definitely be a success (it means we are done). In the other case, we managed to take a variable out of the basis and replace it with a new one. But, more importantly, we improved the objective function by $\theta_*\bar{c}_k$, which would be a smaller success (but a success nonetheless). I say "would be" because if (for some reason) $\theta_*\bar{c}_k = 0$, then we didn't actually improve the objective function at all. So we still have some issues to deal with — in particular:

1. What do we do if none of the $\bar{c}_k$ improve the objective?

2. What happens if an $\bar{c}_k$ does improve the objective function, but the resulting $\theta_* = 0$?

The first situation is easy to deal with because of the following theorem, which we will prove on Problem Set 4:

Theorem 14. *Let* $\mathbf{x}$ *be a basic feasible solution of a linear program* P *for which none of the reduced costs improve the objective function. That* $\mathbf{x}$ *is an optimal solution for* P*.*

Note that "not improve" includes the possibility that some of the $\bar{c}_j = 0$ (for example). The second situation, however, is a bit more touchy.

[29] If there happens to be a tie, then we need to find some way to decide which one to put in, but again we can deal with that later.

4.3 Degeneracies (and cycling)

Notice that in order for θ_* to be 0, it means that the variable we chose to take out (say y_t) had the value 0 in our current BFS $\mathbf{y}$. But remember that in equality standard form, having $y_t = 0$ means that this is an active constraint — in particular, we have more active constraints than we need at this BFS. A BFS $\mathbf{y} \in \mathbb{R}^n$ is called *degenerate* if there are more than n constraints that are active at $\mathbf{y}$. In theory, this is not a problem — we could simply run simplex exactly how we would before, move a total distance of $\theta_* = 0$ in some direction (which is another way of saying we just stay at $\mathbf{y}$) and then swap out y_k for y_t in the basis. Even though we didn't move, we didn't "do nothing" because we now have a new basis and this will give us new edges we can try to use to improve our objective function.

In practice, however, this can be an issue. The reason is that, although we haven't discussed it yet, having an algorithm that improves the objective function at every step is really nice, because it guarantees you never return to the same point twice. [30] If we are allowed to do things that don't improve the objective function, it is possible that we could start going in circles.

For example, imagine you had a basis $\beta = \{y_2, y_3, y_5\}$ and then you do all of this hard work to determine that the best thing to do is to remove y_3 from the basis and replace it with y_1. Now you have a new basis $\beta' = \{y_1, y_2, y_5\}$ and you go and do all of this hard work again to determine that the best thing to do is to remove y_1 from the basis. Guess what might end up replacing it? Since you haven't moved, y_3 is still active (even though it is no longer in the basis) and so it is quite possible that y_1 gets replaced by y_3 which puts you back in the β basis.

4.3.1 Bland's Rule

This phenomenon is called *cycling* and it is something we very much would like to avoid. Now I know what you are thinking — "Hey, I'm a pretty smart person, so I can just look out for these things and make sure it never repeats." But here is the thing: regardless of what you end up doing in your life, there is likely only one time and place that you will need to *actually perform the physical act of "solving a linear program"* and that is in this class. After that, you will want very much to never solve a linear program by hand again and will instead get a computer to do it for you. And the issue with computers is that they are only as smart as the programs running on them, so it would be better if we could find a programmatic way to make sure that this doesn't happen. One popular method for accomplishing this is known as *Bland's rule* or *smallest index rule*:

1. If there are multiple variables that, when added to the basis, would improve the objective function, always choose to add the one with the smallest index.

2. If there are multiple variables that tie for the minimum value θ_*, always choose to remove the one with the smallest index.

It is known that Bland's rule avoids cycling[31] and is actually quite nice in a number of respects:

- It does not require you to keep track of any of the previous bases (like your "I'm a pretty smart person" method probably needed)

- It simplifies computations in the respect that you don't have to calculate reduced costs for every nonbasic variable — just do them in order until you find one that will improve the objective function, and then use that one.

[30] This would be like an M. C. Escher drawing where you go up 4 sets of stairs and end up back where you were (which, fortunately, only happens in M. C. Escher paintings).

[31] We will not prove this in class, but you can assume it is true for the purpose of this class.

4.3.2 Kick the polyhedron!

There is a second "clever" way to get rid of cycling, and that is to add a small amount of random noise to $\mathbf{b}$, which I like to think of this as kicking the polyhedron to cause all of these constraints which magically line up to no longer line up). The idea is that as long as noise is small enough, then you shouldn't change which BFS is the optimal solution. We saw something like this before [32] when we were dealing with the situation of having a non-unique optimal solution — in that case, the "fix" was to add random noise to $\mathbf{c}$. The difference there was that changing $\mathbf{c}$ did not change the feasible region at all, so it was easy to translate solutions back and forth. This time, we are changing the polyhedron, so how are we going to calculate the new point?

Well, keep in mind that "points" aren't really the central object in our algorithm — *bases* are. And bases are much more "stable" than points are. To change a basis, you have to do something extreme — take something out and put something else in. To change a point, you just need to add a little noise. So the fact that everything happens in the "basis" language is actually quite useful in this scenario (and we will seen more advantages later). So the idea would be to solve the new linear program, get an *optimal basis*, and then move that basis back to the old linear program. Given a linear program

$$\min \{\mathbf{c} \cdot \mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$$

a feasible basis β is called an *optimal basis* if the matrix $\mathbf{B} = \text{col}_\beta(\mathbf{A})$ satisfies

- $\mathbf{B}^{-1}\mathbf{b} \geq \mathbf{0}$ (this is needed for feasibility)
- $\mathbf{c}^\intercal - \mathbf{c}_\beta^\intercal \mathbf{B}^{-1}\mathbf{A} \geq \mathbf{0}$ (see Theorem 14).

As long as you don't move things too much, BFS's should map to BFS's so in the end you can simply calculate the point in the old LP from the same optimal basis. This brings us to one final observation — we still can't solve linear programs. The algorithm we came up with here *could* solve linear programs, assuming we had a way to find a BFS to start it on. Fortunately, we will see a method for finding a BFS of a polyhedron on the problem set.

4.3.3 References

1. Bertsimas and Tsitsiklis: Section 2.4, 3.1–3.2

[32] Computer science is full of scenarios that are potentially problematic, but only if some very special conditions happen. Adding small (but random) noise is a useful technique to eliminate the need to deal with such scenarios.

5 Week 5: LP Duality

If you haven't read Section 3.1.1 (with excellent commentary!), you should read it. Specifically, read them for the commentary. The reason I say this is that the commentary points out what is a fundamental imbalance in the world of mathematics — the difference between something existing and something not existing.

5.1 Certificates

On the surface, existence vs. non-existence seems like an even match. Some things exist, some don't. Yes, or no. Yin and Yang (etc). The issue comes when you start trying to *prove* things. Let's go back to the optimization problem from the very first class:

$$\max f(x) \quad \text{such that} \quad x \in X$$

where $f(x)$ is the "height" function and X is the set of people in our class. As we noted, this is an annoying problem because (in general) the only way to solve it is to ask every single person in the class how tall they are (that is, calculate $f(x)$ for every feasible x) and then to find the maximum. But let's say I did all the work and went around and asked everyone and *found the actual maximum*: 224 cm.

So then when the Dean comes in and asks me the solution to my optimization problem, I tell him "224 cm." But how can he check to see that my answer is right? It would be nice to be able to prove to him (somehow) that my answer is correct, without having to force him to do the problem all over again. Such a thing is called a *certificate*. It is a way to show that an answer is correct without going through the effort of solving the entire problem[33] If you have any experience with computational complexity, the class of problems "NP" contains precisely those problems that have certificates (despite those certificates possibly being hard to find).

The truth is, there is no way to provide a certificate showing that 224 cm is the tallest person in the class. At least, not completely. In order to *prove* that 224 cm is the tallest height in the class, I really need to prove two things:

1. There is a person in the class that is 224 cm tall
2. Nobody in the class is taller than 224 cm

The first part *does* have a certificate — to prove that someone in the class is 224 cm tall, I simply point to Andre the Giant (who, in case you didn't know, is an honorary member of the class). Then the Dean can go over and measure Andre and see that he is 224 cm tall (at least according to his biography) and now he can be confident that the first statement is true (he has seen proof). This is where the difference between existence and non-existence becomes clear. If you can find something, proving that thing exists is easy (you just show everyone the thing that you found). On the other hand, how do you prove something *doesn't* exist? Well, one way would be to go and ask every single person in the class their height again (which is as hard as solving the original problem). Are there other ways?

[33] This should suggest that the process of finding a certificate is at least as hard as solving the problem in the first place.

5.1.1 Certificates for Linear Programs

In general, no,[34] but in some (very special) cases, yes, and the case of linear programming is one of those cases[35]. Let's say I have the linear program (in inequality standard form)

$$\begin{aligned} \min\ & x\ + y \\ \text{s.t. } & 2x + 3y \geq 4 \\ & x + 2y \geq 3 \\ & x\ \ \ \ \ \ \geq 0 \\ & \ \ \ \ \ \ y \geq 0 \end{aligned}$$

And let's say I have taken this course already and know how to find the answer and the answer I get is 2. Now to show you my answer is right, I need to show you that (if we let s be the true solution)

1. $s \leq 2$, and
2. $s \geq 2$

Showing that $s \leq 2$ is easy — I just show you the point $(1, 1)$. Now you can check that this point satisfies the constraints (it does) and then you can calculate the value of the objective function at that point (it is 2), so the true minimum is definitely ≤ 2.

To show that $s \geq 2$, what I will try to do is to generate new inequalities from the old ones by combining them in different ways. For example, $x + 2y \geq 3$ and $y \geq 0$ when added together gives $x + 3y \geq 3$ (a new inequality!). Of course this is not a particularly helpful inequality for us, since we are interested in $x+y$ and not $x+3y$. So we will try to be a bit more clever and find a way to write the objective covector $\mathbf{c}^\intercal = [1, 1]$ as a combination of the constraint covectors

$$\text{row}_1(\mathbf{A}) = [2, 3] \quad \text{row}_2(\mathbf{A}) = [1, 2] \quad \text{row}_3(\mathbf{A}) = [1, 0] \quad \text{row}_4(\mathbf{A}) = [0, 1].$$

This is equivalent to multiplying $\mathbf{Ax} \geq \mathbf{b}$ on the left by a covector. For example, if I take $\mathbf{u}^\intercal = [1/3, 0, 1/3, 0]$ then

$$\mathbf{Ax} \geq \mathbf{b} \quad \Rightarrow \quad \mathbf{u}^\intercal\mathbf{Ax} \geq \mathbf{u}^\intercal\mathbf{b}$$

which, when I multiply it all out says $x + y \geq 4/3$, which is a nice lower bound, but not the one I was hoping for (to prove what I want, I would need to get $x + y \geq 2$). But I can fix this by instead using the covector $\mathbf{u}^\intercal = [2, -2, -1, -1]$, so that when I simplify $\mathbf{u}^\intercal\mathbf{Ax} \geq \mathbf{u}^\intercal\mathbf{b}$, I get $x + y \geq 2$, which then proves my answer is right.

Yes, yes, I am sure many of you are saying "but that's not how inequalities work" and the reason you might be saying that is because you are correct: that is not how inequalities work. This is one of the reasons that inequalities tend to be harder to work with than equalities — you can add/subtract/multiply/etc equalities without any issues, but inequalities are not so easy. When we multiply the second constraint by -2 we get

$$-2x - 4y \leq -6$$

which leaves me with a $\geq$ and $\leq$ and I can't add those together like I would a normal equation.

HOWEVER, this general idea would work as long as I add *nonnegative* multiples of the constraints together. In particular, anytime I can find a nonnegative linear combination of

[34] Although it is not clear what "in general" means here. This is related to an extremely important (as of yet unsolved) problem in computational complexity concerning whether NP is the same as co-NP.

[35] This is one of the reasons that linear programming is such a useful optimization tool.

constraints that add up to $[1, 1]$, that will give me a lower bound on the solution to the problem (like $\hat{u} = [1, 1, 0, 0]$ that we tried above). So, for example, I could try $\hat{u} = [1/5, 1/5, 2/5, 0]$ which will give me the new inequality

$$x + y \geq \frac{7}{5}$$

which is slightly better than $4/3$ but not good enough to prove what I want to prove: that $x + y \geq 2$. The goal then would be to look for a nonnegative covector $\hat{u}$ such that

$$\mathbf{u}^\intercal \mathbf{A} = [1, 1] \quad \text{and} \quad \mathbf{u}^\intercal \mathbf{b} = 2$$

Such a covector $\hat{u}$ would then provide a certificate that, in fact, $s \geq 2$ and so my answer would be correct. As it turns out, such a covector $\hat{u}$ doesn't exist, and this is not hard to see. When we combine constraints together correctly, points that used to be feasible should not suddenly become infeasible. So when I see that the point $(x, y) = (1, 1/2)$ is feasible in the initial linear program with value $3/2$, I should be very suspicious if I combined constraints in some way and got $x + y \geq 2$ because this new constraint would make $(1, 1/2)$ infeasible.

In fact, $(x, y) = (1, 1/2)$ is the true solution to this problem, and this time we *can* find a certificate that will allow us to prove it: $\mathbf{u}^\intercal = [0, 1/2, 1/2, 0]$ gives

$$\mathbf{u}^\intercal \mathbf{A} = [1, 1] \quad \text{and} \quad \mathbf{u}^\intercal \mathbf{b} = 3/2.$$

Hence I can use the certificate $(x, y) = (1, 1/2)$ to prove $s \leq 3/2$ and the certificate $\mathbf{u}^\intercal = [0, 1/2, 1/2, 0]$ to prove $s \geq 3/2$. Most importantly, however, both certificates can be checked with only basic linear algebra knowledge (no knowledge of how we actually found these certificates is needed).

This second certificate is called a *dual* certificate and while it may (at this point) look like we just came up with this idea for making dual certificates out of nowhere, it is actually nothing new at all[36]. But to actually prove that your solution to the problem is correct, you have to find one that matches the solution you got! Fortunately, I was able to find such a certificate for the problem above, but it is not clear whether a certificate this good always exists (though we will return to this question in a moment).

In conclusion, it is one thing to be able to solve an optimization problem, but it is a completely different thing to be able to prove to someone what the right answer is without having to re-solve the entire problem[37] In general, proving one side of an inequality is easy (you just find a feasible solution) whereas showing the other side of the inequality is difficult (because it is hard to prove the lack of the existence of something). Linear programming, however, comes with a "built in" way to get dual certificates and each such certificate will give a bound (of different quality) on the difficult side. And if we can find a certificate and a dual certificate that have matching values, this makes it easy to prove that a given solution is correct. So there are really three options at this point:

1. This certificate-finding method will always be able to produce an "optimal certificate" that matches the optimal value of the linear program.

2. All linear programs have an "optimal certificate" but this method will not always work to find one

[36] Let's put our detective hats on and see what is happening here: the dual certificate we are looking for is a way to write the objective vector as a nonnegative combination of constraint vectors — is this similar to something we have seen before (cough, cough, Lagrange Multipliers, cough, cough)?

[37] And, frankly, certificates are *better* than solving the whole problem again, because if you made a mistake the first time, chances are good that you will make the same mistake the second time.

3. There are some linear programs which do not have "optimal certificates"

Obviously the first option would be the best (for us), so let's see if it is true.

5.1.2 Good Certificates for Linear Programs!

We have seen that if we have a linear program

$$\begin{aligned} \mathcal{P} = \min \quad & \mathbf{c} \cdot \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \end{aligned}$$

then any $\mathbf{u} \geq \mathbf{0}$ for which $\mathbf{u}^\intercal \mathbf{A} = \mathbf{c}^\intercal$ will give you a certificate (which can be used to prove a lower bound on the optimal solution), and the quality of that certificate depends on the value of $\mathbf{u}^\intercal \mathbf{b}$. So if we are going to spend time looking for a certificate, we might as well try to find a good one. How good could we do? Well if we were to write down a characterization of the "best possible" certificate (of this type), it would be something like

$$\begin{aligned} \mathcal{D} = \max \quad & \mathbf{b} \cdot \boldsymbol{\lambda} \\ \text{s.t.} \quad & \boldsymbol{\lambda}^\intercal \mathbf{A} = \mathbf{c}^\intercal \\ & \boldsymbol{\lambda} \geq \mathbf{0} \end{aligned} \tag{13}$$

which is a LINEAR PROGRAM! We call this new linear program the *dual* of the original one, and I claim that this is an extremely interesting phenomenon (which we will start to examine in the next section).

5.1.3 References

1. Bertsimas and Tsitsiklis: Section 4.1

5.2 Duality

The goal now is to formalize this notion the we saw in the previous section and generalize it to situations where we are not (necessarily) in inequality standard form. But before doing that, take another look at what (13) is trying to do... it is trying to write the objective function as a linear combination of the constraints, which is exactly how Lagrange multipliers work. Fortunately, we developed a geometric intuition of Lagrange multipliers (Section 1.2.2) and so we can expect this to come in handy when trying to understand this strange new "duality" concept.

5.2.1 Dual linear programs

There is one small point of language that we should clear up before moving on. Recall that we had the linear programs

$$\begin{aligned} \mathcal{P} = \min \quad & \mathbf{c} \cdot \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \end{aligned} \qquad\qquad \begin{aligned} \mathcal{D} = \max \quad & \mathbf{b} \cdot \boldsymbol{\lambda} \\ \text{s.t.} \quad & \mathbf{A}^\intercal \boldsymbol{\lambda} = \mathbf{c} \\ & \boldsymbol{\lambda} \geq \mathbf{0} \end{aligned}$$

above and we said that $\mathcal{D}$ is called the *dual* of $\mathcal{P}$. In a moment, we will define a duality operator that can be applied to any linear program to form "the dual" and it is quite common in the literature to see the original linear program being called "the primal." However we will see that

this duality operator is an *involution* (that is, applying it twice gets you back to where you started, like the function $1/x$). So I find it a bit misleading to call $\mathcal{P}$ the "primal" because there was nothing special about $\mathcal{P}$ apart from the fact that it was the linear program I started with. Had I started with $\mathcal{D}$, then I could very easily be calling it the "primal" and $\mathcal{P}$ the "dual" and one can see how this would get confusing. So instead I will refer to $\mathcal{P}$ and $\mathcal{D}$ as a *primal/dual pair*.

The most important property of $\mathcal{P}$ and $\mathcal{D}$ that we will want to preserve in general primal/dual pairs is that for any $\mathbf{x} \in \text{feas}(\mathcal{P})$ and $\boldsymbol{\lambda} \in \text{feas}(\mathcal{D})$, we want to have

$$\mathbf{c} \cdot \mathbf{x} \geq \mathbf{b} \cdot \boldsymbol{\lambda}. \tag{14}$$

That way any feasible solution for $\mathcal{P}$ serves as a certificate for $\mathcal{D}$ and any feasible solution to $\mathcal{D}$ serves as a certificate for $\mathcal{P}$. So, in particular, if you could find a $\mathbf{x}^* \in \text{feas}(\mathcal{P})$ and $\boldsymbol{\lambda}^* \in \text{feas}(\mathcal{D})$ for which

$$\mathbf{c} \cdot \mathbf{x}^* = \mathbf{b} \cdot \boldsymbol{\lambda}^*$$

then $\mathbf{x}^*$ and $\boldsymbol{\lambda}^*$ would have to be optimal solutions to $\mathcal{P}$ and $\mathcal{D}$ (respectively).

To find the "dual" of an arbitrary linear program, we first turn it into a minimization problem and then use the map:

$$\begin{array}{rcl}
\min \ \mathbf{c} \cdot \mathbf{x} & \Longrightarrow & \max \ \boldsymbol{\lambda} \cdot \mathbf{b} \\
\text{row}_i(\mathbf{A}) \cdot \mathbf{x} \geq b_i & \Longrightarrow & \lambda_i \geq 0 \\
\text{row}_i(\mathbf{A}) \cdot \mathbf{x} \leq b_i & \Longrightarrow & \lambda_i \leq 0 \\
\text{row}_i(\mathbf{A}) \cdot \mathbf{x} = b_i & \Longrightarrow & \lambda_i \text{ free} \\
x_j \geq 0 & \Longrightarrow & \text{col}_j(\mathbf{A}) \cdot \boldsymbol{\lambda} \leq c_j \\
x_j \leq 0 & \Longrightarrow & \text{col}_j(\mathbf{A}) \cdot \boldsymbol{\lambda} \geq c_j \\
x_j \text{ free} & \Longrightarrow & \text{col}_j(\mathbf{A}) \cdot \boldsymbol{\lambda} = c_j
\end{array}$$

Figure 1: The duality map for general LP constraints.

The first property that one can check is the one mentioned above: that this map is an *involution*. That is, if we started with a minimization problem $\mathcal{P}$ and

- Found $\mathcal{D}$ (a maximization problem) using the duality map above
- Turned $\mathcal{D}$ into a minimization problem $\mathcal{D}'$ by multiplying the cost by -1
- Did the duality map above to $\mathcal{D}'$ to get a new maximization problem $\mathcal{Q}$
- Turned $\mathcal{Q}$ into a minimization problem $\mathcal{Q}'$ by multiplying the cost by -1

then we would find that $\mathcal{Q}' = \mathcal{P}$. Not that $\mathcal{Q}'$ and $\mathcal{P}$ are equivalent — they would be *exactly the same*. So this tells us that the arrows in the duality map are actually double arrows, and so we can move back and forth between a minimization problem and its dual maximization problem by going in either direction. So (as we suggested) neither one is really "the primal" or "the dual" — you get to say which is the primal and then the other one becomes the dual.

However if you look closely, going from left to right is DIFFERENT than going from right to left. So how do you know which one to do? That depends entirely on whether you are moving

from min to max (in which case you use $\Rightarrow$) or from max to min (in which case you use $\Leftarrow$). So for example,

$$\begin{array}{rl} \min & \mathbf{c}\cdot\mathbf{x} \\ \text{s.t.} & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \qquad\qquad \begin{array}{rl} \max & \mathbf{b}\cdot\boldsymbol{\lambda} \\ \text{s.t.} & \mathbf{A}^\intercal\boldsymbol{\lambda} \leq \mathbf{c} \end{array}$$

and

$$\begin{array}{rrcrcrcl} \min & x_1 & + & 2x_2 & + & 3x_3 & & \\ \text{s.t.} & -x_1 & + & 3x_2 & & & = & 5 \\ & 2x_1 & - & x_2 & + & 3x_3 & \geq & 6 \\ & & & & & 2x_3 & \leq & 7 \\ & x_1 & & & & & \geq & 0 \\ & & & x_2 & & & \leq & 0 \\ & & & & & x_3 & \text{free} & \end{array} \qquad \begin{array}{rrcrcrcl} \max & 5\lambda_1 & + & 6\lambda_2 & + & 7\lambda_3 & & \\ \text{s.t.} & -\lambda_1 & + & 2\lambda_2 & & & \leq & 1 \\ & 3\lambda_1 & - & \lambda_2 & & & \geq & 2 \\ & & & 3\lambda_2 & + & 2\lambda_3 & = & 3 \\ & \lambda_1 & & & & & \text{free} & \\ & & & \lambda_2 & & & \geq & 0 \\ & & & & & \lambda_3 & \leq & 0 \end{array}$$

are both primal/dual pairs. The key property that the duality map enforces is that

$$\lambda_i(\text{row}_i(\mathbf{A})\cdot\mathbf{x} - b_i) \geq 0 \quad \text{and} \quad x_j(c_j - \text{col}_j(\mathbf{A})\cdot\boldsymbol{\lambda}) \geq 0$$

for all i, j because that is what ensures that we get the same inequality as in (14):

Theorem 15 (Weak Duality)**.** *Let $\mathcal{P}$ be a minimization problem and $\mathcal{D}$ a maximization problem related via the map in Figure 1. Then for all $\mathbf{x} \in \text{feas}(\mathcal{P})$ and all $\boldsymbol{\lambda} \in \text{feas}(\mathcal{D})$ we have*

$$\mathbf{b}\cdot\boldsymbol{\lambda} \leq \mathbf{c}\cdot\mathbf{x}$$

Or, in other words, a feasible solution to the dual acts as a certificate for the primal (and vice versa). So the big question now is: can I always find a certificate that matches the optimal value of the primal?

5.2.2 Strong Duality

To investigate this, we need to look at what the dual problem is doing — trying to find a nonnegative linear combination of the constraints that added up to the objective function. That sounds like... Lagrange Multipliers (see Section 1.2.2)! Lagrange multipliers tells us that at the optimal solution, the objective function needs to be a nonnegative linear combination of the active constraints. In the language of simplex, that becomes Theorem 14 which in turn gives the following theorem

Theorem 16 (Strong Duality)**.** *If a linear program has a finite optimal solution, then it's dual has a finite optimal solution and the two optimal values are equal.*

This is where we see that the idea of a "basis" actually works quite well with this whole duality thing. In particular, if I am doing simplex to an equality standard form problem and I have a basis β, then that basis gives me a solution in the original problem:

$$\mathbf{x}_\beta = \mathbf{B}^{-1}\mathbf{b} \qquad \text{and 0 elsewhere}$$

but it also gives me a solution in the dual:

$$\boldsymbol{\lambda}^\intercal = \mathbf{c}_\beta^\intercal\mathbf{B}^{-1}$$

and since the matrix goes from $\mathbf{A}$ to $\mathbf{A}^\intercal$ in the dual map, the column basis gets mapped to a row basis. As a result, an equivalent way of defining optimal basis would be "any basis for which both associated solutions are feasible."

Note that Theorem 16 only considers the case of finite optimal solutions. What happens when $\mathcal{P}$ (and/or $\mathcal{D}$) is infeasible or unbounded is still something to decide (see Problem Set 5).

On the other hand, if we think back to the physical meaning of Lagrange multipliers, I claim we can see something even stronger than the fact that the two linear programs have matching solutions[38]. It told us something about the structure of the solutions — that the only constraints that should have positive weight are the ones that are active. This is known as *complementary slackness*.

Theorem 17. *Let $\mathcal{P}$ be a minimization linear program with feasible solution $\mathbf{x}$, and let $\mathcal{D}$ be its dual (maximization) linear program with feasible solution $\boldsymbol{\lambda}$. Then $\mathbf{x}$ and $\boldsymbol{\lambda}$ are optimal solutions if and only if*

$$\begin{aligned} \lambda_i(\text{row}_i(\mathbf{A}) \cdot \mathbf{x} - b_i) &= 0 \qquad \textit{for all } i \\ x_j(\text{col}_j(\mathbf{A}) \cdot \boldsymbol{\lambda} - c_j) &= 0 \qquad \textit{for all } j \end{aligned}$$

In other words,

- $\lambda_i = 0$ whenever constraint i is non-active in $\mathcal{P}$
- $x_j = 0$ whenever constraint j is non-active in $\mathcal{D}$.

The proof of this uses the idea from the previous section that if you add up a bunch of nonnegative things and get 0, it means each of the individual entries must be 0.

5.2.3 References

1. Bertsimas and Tsitsiklis: Section 4.1–4.3

[38] See also Example 4.4 in the book.

6 Week 6: Applications of Duality

6.1 Farkas' Lemma

Our first application of LP duality is known as *Farkas' Lemma*. There are two remarks that should be made:

- the word *application* is a bit misleading because it is generally considered to be equivalent to strong duality (that is, you can use either one to prove the other).
- the word *lemma* is misleading because it is generally considered to be a collection of lemmas.

We saw an example of a Farkas Lemma in the problem set:

Lemma 18 (Farkas)**.** *For an $m \times n$ matrix $\mathbf{A}$, exactly one of the following holds:*

1. *There exists a vector $\mathbf{x} \geq \mathbf{0}$ with $\mathbf{Ax} = \mathbf{b}$*
2. *There exists a vector $\boldsymbol{\lambda}$ such that $\boldsymbol{\lambda}^\intercal \mathbf{A} \geq \mathbf{0}$ and $\mathbf{b} \cdot \boldsymbol{\lambda} < 0$.*

Theorems of the type "exactly one can happen" are called *theorems of the alternative*, but really it is just a modified if and only if, since $A \; XOR \; B$ is equivalent to $A \iff (NOT \; B)$.

Another one that we proved in class was:

Lemma 19 (Farkas)**.** *For vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m, \mathbf{b} \in \mathbb{R}^m$, exactly one of the following holds:*

1. *There exists a vector $\mathbf{x} \geq \mathbf{0}$ satisfying $\mathbf{b} \cdot \mathbf{x} \geq \max_i \{\mathbf{a}_i \cdot \mathbf{x}\}$*
2. *There exists a vector $\boldsymbol{\lambda} \geq \mathbf{0}$ such that $\boldsymbol{\lambda} \cdot \mathbf{1} = 1$ and $\mathbf{b} \leq \sum_i \lambda_i \mathbf{a}_i$.*

So what is a "Farkas Lemma" exactly — I think it depends on who you ask, but typically it is anything that gives a duality-type statement without really mentioning duality (we knew enough to understand the statements of these lemmas the first week of class, but didn't know enough to prove them until the previous chapter).

6.1.1 Separating hyperplanes

As we have mentioned, the reason we like "theorems of the alternative" is that it gives a pair of complementary certificates. Either A happens and B doesn't (in which case A provides a certificate) or B happens and A doesn't (in which case B provides a certificate). One advantage of removing the "duality" framework is that it is often easier to generalize. For example, we saw on the problem set the idea of a separating hyperplane. Given two sets X, Y, we say they are *separated by a hyperplane* if there exists a vector $\mathbf{v}$ and scalar c for which

$$\mathbf{x} \cdot \mathbf{v} \geq c \quad \text{for all } x \in X \quad \text{and} \quad \mathbf{y} \cdot \mathbf{v} < c \quad \text{for all } y \in y$$

The existence of a separating hyperplane is a certificate that $X \cap Y = \varnothing$ (and vice versa). One consequence of strong duality is the following lemma:

Lemma 20 (Separating hyperplanes)**.** *Given a point $\mathbf{x}$ and polyhedron P, exactly one of the following holds:*

1. $\mathbf{x} \in P$

2. $\mathbf{x}$ *and* P *can be separated by a hyperplane*

As we saw in class, however, this can be generalized from polyhedra to "all convex sets Y." The proof started by examining the situation when Y was a single point (this was easy) and then considered the general case by finding a point $\mathbf{y} \in Y$ that minimized the function $\|\mathbf{x}-\mathbf{y}\|^2$ (using Weierstraß's theorem on compact sets). I then claimed that any hyperplane which separated $\mathbf{x}$ from $\mathbf{y}$ also separated $\mathbf{x}$ from the rest of Y. This was a consequence (I claimed) of convexity and the triangle inequality, which of course can apply in much more general settings than polyhedra in $\mathbb{R}^n$, so this is why knowing that these things exist outside of LP duality is useful.

6.1.2 References

1. Bertsimas and Tsitsiklis: Section 4.6

6.2 Games: minimax and maximin

Another application of duality comes from an economics perspective, which tends to have a different theme than in optimization. As opposed to our way of approaching it (as two LP's that are in harmony), economics sees duality as two agents that are fighting against each other (like a buyer and a seller) to get the best price for some goods. This type of situation is often thought of as 2 players competing in a "game".

By "game", I mean an idealized, one round, no information game which we can model using a (payout) function $f(x, y)$ and the "game" consists of player X picking a value x and player Y picking a value y and then Y gives X a total of $f(x, y)$ francs/dollars/clams/whatever. So player X wants to pick a x that makes $f(x, y)$ as big as possible and Y wants to pick a y to make $f(x, y)$ as small as possible and both must decide what to pick without seeing the other's choice. We saw that there were two natural quantities associated with such games:[39]

- the *maximin* value: if Y were able to see x (the value X picked) before picking y, this is the best outcome X could force.

$$\underline{f} = \max_{x \in X} \min_{y \in Y} f(x, y)$$

- the *minimax* value: if X were able to see y (the value Y picked) before picking x, this is the best outcome Y could force.

$$\overline{f} = \min_{y \in Y} \max_{x \in X} f(x, y)$$

We saw in class that, for all games, $\underline{f} \leq \overline{f}$ and that (in general) they are not equal. The interpretation we came up with was that $\overline{f} - \underline{f}$ was the value of "knowledge" in the game — how much do you gain by knowing what your opponent is playing before you decide?

6.3 Mixed Games

In general, games like the one above are quite hard (and may not have a fixed solution). So instead we often consider a modified version of these "games" called a *mixed game* which works as follows:

- X and Y each has a finite set of moves (just like before)

[39] Note that the way to read these is that the playing goes from left to right. That is, $\max_{x \in X} \min_{y \in Y} f(x, y)$ has X picking x first, Y knowing what y picked, and then Y getting to pick y using that knowledge.

- instead of picking one move each, they pick a probability distribution over their possible moves
- a neutral referee uses a random number generator to determine which moves X and Y do based on these distributions, and then the payout is calculated.

Mathematically, we can define the set of probability distributions over k possible choices as

$$\Delta_k = \left\{ \mathbf{x} \in \mathbb{R}^k : \sum_i x_i = 1, \mathbf{x} \geq 0 \right\}$$

and the function $f_{\mathbf{M}}$ could be written as a matrix (where the rows correspond to moves that X can make and the columns correspond to moves Y can make). And this was nice because it gave a nice compact way to see what the expected value of the game was:

$$\mathbb{E} f_{\mathbf{M}}(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\intercal \mathbf{M} \mathbf{y}$$

where the expectation is over the random number generator. The goals of X and Y remain the same, only now they want to maximize/minimize the expected payout.

6.3.1 The minimax theorem (for mixed games)

The main theorem we proved was the following:

Theorem 21 (Von Neumann)**.** *For any $m \times n$ payout matrix M, we have*

$$\underline{f_{\mathbf{M}}} = \max_{\mathbf{x} \in \Delta_m} \min_{\mathbf{y} \in \Delta_n} \mathbf{x}^\intercal \mathbf{M} \mathbf{y} = \min_{\mathbf{y} \in \Delta_n} \max_{\mathbf{x} \in \Delta_m} \mathbf{x}^\intercal \mathbf{M} \mathbf{y} = \overline{f_{\mathbf{M}}}.$$

We took this to mean that

1. The mixed game $f_{\mathbf{M}}$ has a unique "value" $T = \underline{f_{\mathbf{M}}} = \overline{f_{\mathbf{M}}}$
2. X and Y each have an "optimal strategy" which would guarantee them a value of T regardless of whether the opponent knew their strategy.

The proof uses strong duality: we first write a linear program that would give X a good strategy:

$$\begin{array}{lrcl} \max & t & & \\ \text{s.t.} & \mathbf{x}^\intercal \mathbf{M} \quad -t\,\mathbf{1} & \geq & \mathbf{0} \\ & \mathbf{x} \cdot \mathbf{1} & = & 1 \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}.$$

In other words, the optimal solution $(\mathbf{x}_*, t_*)$ satisfies

$$\min_{\mathbf{y} \in \Delta_n} \mathbf{x}_*^\intercal \mathbf{M} \mathbf{y} = t_* \leq \underline{f_{\mathbf{M}}}.$$

But the interesting thing was that when we took the dual of this linear program, we got something that was equivalent to[40] the linear program

$$\begin{array}{lrcl} \min & s & & \\ \text{s.t.} & \mathbf{M}\mathbf{y} \quad -s\,\mathbf{1} & \leq & \mathbf{0} \\ & \mathbf{y} \cdot \mathbf{1} & = & 1 \\ & \mathbf{y} & \geq & \mathbf{0} \end{array}$$

[40] They are not exactly duals of each other, but they will have the same optimal value.

which is the linear program that Y would use to find a good strategy. In other words, the optimal solution to this linear program $(\mathbf{y}_*, s_*)$ satisfies

$$\max_{\mathbf{x}\in\Delta_m} \mathbf{x}^\intercal \mathbf{M}\mathbf{y}_* = s_* \geq \overline{f_{\mathbf{M}}}.$$

So on one hand we have a string of inequalities

$$t_* \leq \underline{f_{\mathbf{M}}} \leq \overline{f_{\mathbf{M}}} \leq s_*$$

and on the other hand, strong duality tells us that $s_* = t_*$. Hence the system of inequalities collapses to a system of equalities:

$$t_* = \underline{f_{\mathbf{M}}} = \overline{f_{\mathbf{M}}} = s_*$$

and so $\mathbf{x}_*$ and $\mathbf{y}_*$ are actually optimal strategies.

6.4 Thinking in terms of quantifiers

If it wasn't clear why the linear programs in the previous section find an optimal solution, it may help to switch from the max min way of describing things to one that uses quantifiers: recall that $\underline{f}$ is the one where Y knows what X is going to play. Hence to say $\underline{f} < t$ is to say that Y always has a response for X that gets a value smaller than t:

$$\underline{f} < t \iff \text{for all } x \in X \text{ there exists a } y \in Y \text{ for which } f(x,y) < t$$

The negation of this statement is

$$\underline{f} \geq t \iff \text{there exists } x \in X \text{ s.t. for all } y \in Y \text{ we have } f(x,y) \geq t$$

So in the mixed game scenario, saying $\underline{f_{\mathbf{M}}} \geq t$ is saying that there exists a vector $\mathbf{x} \in \Delta_n$ for which

$$\mathbf{x}^\intercal \mathbf{M}\mathbf{y} \geq t$$

for all $\mathbf{y} \in \Delta_m$. But $\mathbf{y}$ has the option of putting a 1 in any index that it wants (and 0 in the others). Hence a given $\mathbf{x}$ satisfies

$$\mathbf{x}^\intercal \mathbf{M}\mathbf{y} \geq t \quad \text{for all } \mathbf{y} \in \Delta_m \quad \iff \quad \mathbf{x}^\intercal \mathbf{M} \geq t\mathbf{1}.$$

Putting this all together, $\underline{f_{\mathbf{M}}} \geq t$ if and only if there exists $\mathbf{x} \in \Delta_n$ for which $\mathbf{x}^\intercal \mathbf{M} \geq t\mathbf{1}$. However (by definition) $\underline{f_{\mathbf{M}}}$ is the maximum of all such t, so that is why our linear program has the form that it does.

I mention this primarily because it can often be a useful trick to switch back and forth between the language of max and min and the language of $\exists$ and $\forall$. In particular, one can think about the difference between weak and strong duality using quantifiers. Weak duality is similar to the quantifier statement

$$\exists x \forall y \; A(x,y) \Rightarrow \forall y \exists x \; A(x,y).$$

and it should be easy to see why this will always be true (whatever special x works for the left hand side will still work on the right hand side). Strong duality, on the other hand, is similar to the quantifier statement

$$\exists x \forall y \; A(x,y) \iff \forall y \exists x \; A(x,y).$$

and it should also be clear that (in general) this is NOT true — the left hand side requires a single x to do the job, whereas the right hand side can alter its values of x to match any particular y. And so this is why we use the name "optimal strategy" — it is a single strategy that works uniformly well on all $\mathbf{y}$.

And in some sense, this is the real answer to the question "What makes something a Farkas Lemma?" While moving from optimization language to quantifier language may be straightforward, the reverse direction is less obvious. And so a good way to think about the name "Farkas Lemma" is as a way to label results that use this type of proof technique: given a quantifier statement, turn it into an optimization statement and then use optimization techniques to prove it.

6.4.1 References:

1. Problem sets 6 and 7

2. The topic of mixed games is not something that is super important to the class material — it is an interesting application that I wanted to show you so that you can get some appreciation of how duality (and linear programming) are used in real life. For those interested in a bit learning more about the subject, I can recommend Chapters 9 & 10 in `http://www.ru.ac.bd/wp-content/uploads/sites/25/2019/03/405_01_Thie_An_Introduction_to-linear-programming-and-game-theory.pdf`

3. In fact, optimization has central importance in many areas of game theory (and economics in general). For a much more comprehensive list of topics in this area, I recommend `https://homes.cs.washington.edu/~karlin/GameTheoryBook.pdf`

7 Week 7: Integer Programming

7.1 Complexity of Linear Programming

As far as complexity goes, the most time-consuming operations that are needed to do simplex are dealing with the matrices. In general, for an $m \times n$ matrix, most linear algebra operations (inversion, finding eigenvalues, row reducing, etc) are $O((m+n)^3)$. This can actually be improved when it comes to doing simplex, since there is a way to "update" many of the quantities without having to start over. For example, if you already know $\mathbf{B}_\beta^{-1}$ and then you do one step of simplex to get a new basis β', there is a way to compute $\mathbf{B}_{\beta'}^{-1}$ from $\mathbf{B}_\beta^{-1}$ using $O(n^2)$ steps[41]

What really determines how long it takes for simplex to finish, then, is how many steps it needs to find the optimal solution. If we pick any reasonable way of doing simplex (Bland's rule, for example), it turns out that one can find examples where the number of simplex steps is exponential. And "exponential" is not good (at least as far as algorithms are concerned, so this is one of the (potential) down sides of simplex.

HOWEVER, it turns out the problem of "solving a linear program" does have a polynomial time algorithm (it just isn't simplex). Instead there is an algorithm that uses what is known as the *ellipsoid method* and it has been shown that this algorithm always takes a polynomial amount of time to find an optimal solution[42]. However, the ellipsoid method is not used in practice because the complexity is on the order of $(m+n)^6$ which (in the real world) is big enough that it might as well be exponential (see Section 1.3). So while simplex can sometimes take an exponential amount of time, it has the possibility of taking must shorter time, which is one reason it is used in practice (finishing sometimes is better than never finishing).

It should be mentioned, however, that part of the reason simplex can be forced to take an exponential number of steps is that we never really figured out what the "best" edge to take was (we pick*ed using Bland's rule, which guarantees a good edge, but not necessarily the best one[43]). Pretend, for a moment, that you had a magical fairy that you could use for your pick* subroutine that would always be able to tell you which edge would lead to the shortest possible path to the optimal solution... then (you can ask) would simplex finish in polynomial time? Unfortunately, we don't know. Currently the best known upper bound (due to Kalai and Kleitman) is that two vertices in a polyhedron formed by m constraints in n dimensions always has a path (of edges) between them of length at most $(2n)^{\log m}$ steps (which is not polynomial in m and n if we allow both to grow). Currently the best known lower bound[44] is $m-n+\lfloor n/5 \rfloor$ (so *linear* in m and n).[45] So, somewhat embarrassingly[46], the answer is somewhere between linear and "grows faster than any polynomial" (which is a pretty big gap). But even if the shortest path across a polyhedron was polynomial, it still wouldn't help in practice because we would still need a pick* subroutine that was able to tell us what path to take (and we are nowhere close to finding that, either).

In any case, the important thing to know going forward is that an arbitrary linear programming can be solved in polynomial time. This will have no practical implication whatsoever, but

[41] It is here, if you are interested: `https://en.wikipedia.org/wiki/Sherman-Morrison_formula`.

[42] Technically, you give it a value of ε and it guarantees that it will give you a solution that is within a distance of ε of the optimal solution.

[43] By "best" I mean gives you the shortest possible path from your current vertex to the optimal solution.

[44] In other words, over all possible polyhedra formed from m constraints in n dimensions, and all pairs of vertices on that polyhedra, this is the longest distance anyone has been able to show is necessary to go between those two vertices.

[45] In fact, for a long time people thought that $m-n$ could actually be the length of the shortest path — this was known as the Hirsch conjecture — a counterexample to this was found only fairly recently (by Santos).

[46] This is just one (of many) examples that show how little humans know about the structure of objects in high dimensions.

it is important because in the next section we will start to run into problems for which this is *not* true, and this will help us to understand when certain tasks should be easy and when they should be hard. In particular, the rule of thumb is:

7.1.1 Complexity Rule of Thumb

> If you have a problem that is hard (in the general case), and there exists a method for transforming all possible instances of that problem into something that is known to be easy (in the general case), then you should expect that the transformation is either (1) hard to find, or (2) takes a long time to finish.[47]

7.1.2 References

1. Bertsimas and Tsitsiklis: Chapter 8

7.2 Integer Programming

Integer programming is the same as linear programming, but with extra types of constraints:

1. BIP[48]: binary integer program ($\mathbf{z} \in \{0,1\}^n$)
2. IP: integer program ($\mathbf{z} \in \mathbb{Z}^n$)
3. MILP[49]: mixed integer linear program ($\mathbf{x} \in \mathbb{Z}^n$, $\mathbf{y} \in \mathbb{R}^m$)

All of these are nonconvex!!

$$\text{BIP} \subseteq \text{IP} \subseteq \text{MILP}$$

For example, $x \in \{0,1\}$ is the same as

$$\begin{aligned} x \geq & 0 \\ x \leq & 1 \\ x \in & \mathbb{Z} \end{aligned}$$

In general, BIP and IP are as hard as MILP. Similar to linear programs, every MILP can be written in *equality standard form*:

$$\begin{aligned} \min / \max \ & \mathbf{c} \cdot \mathbf{x} + \mathbf{d} \cdot \mathbf{z} \\ \text{s.t. } & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} = \mathbf{b} \\ & \mathbf{x}, \mathbf{z} \geq 0 \\ & \mathbf{z} \in \mathbb{Z}^n \end{aligned}$$

or *inequality standard form*

$$\begin{aligned} \min / \max \ & \mathbf{c} \cdot \mathbf{x} + \mathbf{d} \cdot \mathbf{z} \\ \text{s.t. } & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} \geq \mathbf{b} \\ & \mathbf{z} \in \mathbb{Z}^n \end{aligned}$$

[47] Because otherwise, this would be (by definition) a way to solve the problem easily.
[48] Sometimes called ZOIP (zero-one integer program)
[49] Sometimes called ILP (integer linear program) or MIP (mixed integer program)

7.2.1 Modeling techniques

Binary constraints are most common (gives you the ability to make choices).

1. Pick at most k out of n:

$$\sum_{i=1}^{n} z_i \leq k$$
$$z_i \in \{0, 1\}$$

   Similar for "at least" ($\geq k$) and "exactly" ($= k$)

2. Dependencies (if z is chosen, w must be chosen):

$$z \leq w$$
$$w, z \in \{0, 1\}$$

3. Restricting a variable to a finite set of values: $u \in \{a_1, \ldots, a_n\}$

$$u - \sum_{i=1}^{n} a_i z_i = 0$$
$$\sum_{i=1}^{n} z_i = 1$$
$$z_i \in \{0, 1\}$$

4. OR constraints: $\mathbf{a}_1 \cdot \mathbf{x} \geq b_1$ or $\mathbf{a}_2 \cdot \mathbf{x} \geq b_2$

$$\mathbf{a}_1 \cdot \mathbf{x} \geq z b_1$$
$$\mathbf{a}_2 \cdot \mathbf{x} \geq (1 - z) b_2$$
$$z \in \{0, 1\}$$

   Only works when $\mathbf{a}_i \geq 0$.

Example 22. To enforce $a \leq |x| \leq b$:

$$x \leq -za + (1 - z)b$$
$$x \geq (1 - z)a - zb$$
$$z \in \{0, 1\}$$

1. when $z = 0$, this forces $a \leq x \leq b$
2. when $z = 1$, this forces $-b \leq x \leq -a$

Not possible in linear programming (not a convex set).

7.2.2 Some examples of integer programs

$(0,1)$-KNAPSACK: Given weight vector $\mathbf{w} \geq 0$ and value vector $\mathbf{v}$

$$\begin{aligned} \max\ & \mathbf{v} \cdot \mathbf{z} \\ \text{s.t. } & \mathbf{w} \cdot \mathbf{z} \leq W \\ & \mathbf{z} \in \{0,1\} \end{aligned}$$

In words: if you have a sack that holds W weight, what items do you pick to maximize your value?

SUBSET-SUM: Given a vector $\mathbf{w}$ and scalar W,

$$\begin{aligned} \max\ & \mathbf{0} \cdot \mathbf{z} \\ \text{s.t. } & \mathbf{w} \cdot \mathbf{z} = W \\ & \mathbf{z} \in \{0,1\} \end{aligned}$$

In words: does there exist a subset of $\{w_1, \ldots, w_n\}$ which sums to W?

Graph chromatic number: Given a graph $G = (V, E)$ with $|V| = n$,

$$\min \sum_{k=1}^{n} y_k \tag{15}$$

$$\text{s.t. } \sum_{k=1}^{n} x_{ik} = 1 \quad \text{for all } v_i \in V \tag{16}$$

$$x_{ik} - y_k \leq 0 \quad \text{for all } v_i \in V \text{ and all } k \tag{17}$$

$$x_{ik} + x_{jk} \leq 1 \quad \text{for all } (v_i, v_j) \in E \tag{18}$$

$$x_{ik},\ y_k \in \{0,1\}$$

In words: find the smallest number of colors needed to color the vertices so that no edge has two ends with the same color. Note that the value of variable y_k tells you whether color k is used (or not) and x_{ik} tells you whether a vertex v_i is given color k (or not). The translation of the integer program is then:

(15) minimize the total number of colors used

(16) each vertex is assigned one color

(17) if any vertex uses color k, set $y_k = 1$ so it is counted in (15)

(18) no two adjacent vertices can be assigned the same color

Maximum weight perfect matching Given a nonnegative $n \times n$ matrix c_{ij}

$$\begin{aligned} \max & \sum_{i,j} c_{ij} x_{ij} \\ \text{s.t. } & \sum_{i} x_{ij} = 1 \quad \text{for all } j \\ & \sum_{j} x_{ij} = 1 \quad \text{for all } i \\ & x_{ij} \in \{0,1\} \end{aligned}$$

In words: given a complete bipartite graph on $2n$ vertices, with weights on the edges, find the perfect matching that has maximum weight.

7.2.3 Geometry (and strong cuts)

For a given optimization problem P, let $\text{feas}(P)$ denote the feasible set of P and $\text{opt}(P)$ the optimal value. Given a mixed integer program

$$\begin{aligned} \mathcal{P}_{\mathbb{Z}} = \min / \max \ & \mathbf{c} \cdot \mathbf{x} + \mathbf{d} \cdot \mathbf{z} \\ \text{s.t. } & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} = \mathbf{b} \\ & \mathbf{x}, \mathbf{z} \geq 0 \\ & \mathbf{z} \in \mathbb{Z}^n \end{aligned}$$

the *LP relaxation* is

$$\begin{aligned} \mathcal{P}_{\mathbb{R}} = \min / \max \ & \mathbf{c} \cdot \mathbf{x} + \mathbf{d} \cdot \mathbf{z} \\ \text{s.t. } & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} = \mathbf{b} \\ & \mathbf{x}, \mathbf{z} \geq 0 \end{aligned}$$

Hence $\text{feas}(\mathcal{P}_{\mathbb{Z}}) \subseteq \text{feas}(\mathcal{P}_{\mathbb{R}}) = Q$ where Q is a polyhedron. As a result, if $\mathcal{P}_{\mathbb{Z}}$ is a maximization problem, then $\text{opt}(\mathcal{P}_{\mathbb{Z}}) \leq \text{opt}(\mathcal{P}_{\mathbb{R}})$. And if $\mathcal{P}_{\mathbb{R}}$ has an optimal solution that has integer coordinates (for any reason), then this is an optimal solution for $P_{\mathbb{Z}}$ as well. However it was clear (looking, for example, at Figure 2) that one could define the feasible set of an integer program in multiple ways — leading to (possibly) many different relaxations. So we defined an equivalence relation on the collection of MILPs: $P \sim P'$ if they have the same objective function and $\text{feas}(P) = \text{feas}(P')$. The different elements in a given equivalence class are called *formulations*. Clearly, if $P \sim P'$ then $\text{opt}(P) = \text{opt}(P')$, but the LP relaxations $\mathcal{P}_{\mathbb{R}}$ and $\mathcal{P}'_{\mathbb{R}}$ can be (very) different. A formulation P is said to be *stronger* than a formulation P' if $\text{feas}(\mathcal{P}_{\mathbb{R}}) \subset \text{feas}(\mathcal{P}'_{\mathbb{R}})$ (notice that these are polytopes).

Finally, we noted that each equivalence class contains a unique "strongest possible" formulation: the formulation $\widetilde{P}$ for which $\text{feas}\,\widetilde{P}_{\mathbb{R}} = \text{conv}(\text{feas}(\widetilde{P}))$. Such a formulation will always satisfy $\text{opt}(\widetilde{P}_{\mathbb{R}}) = \text{opt}(\widetilde{P})$. Furthermore, we can take a given formulation P and turn it into a stronger formulation by adding what are called *cuts (or cutting planes)*. These are new constraints which, when added to P, causes $\text{feas}(\mathcal{P}_{\mathbb{R}})$ to get smaller while keeping $\text{feas}(P)$ constant.[50]

Figure 2 shows a polyhedron P defined by the linear constraints of an integer program (in solid blue). The feasible set S consists of all integer points inside the polyhedron. The optimal solution to the LP relaxation is $\mathbf{x}_{\mathbb{R}}$, and the dashed blue line represents a valid cut that would result in a stronger IP formulation and improve the the solution to the LP relaxation. $\text{conv}(S)$ is shown in red and represents the strongest possible IP formulation. In particular, the optimal solution to the formulation $\widetilde{P}$ for which $\text{feas}(\widetilde{P}_{\mathbb{R}}) = \text{conv}(S)$ will have integer coordinates (and therefore be an optimal solution for the original integer program).

We discussed two "general" ways of finding cuts:

7.2.3.1 Rounding cuts

Consider the constraints

$$\begin{aligned} 2x + y &\geq 1 \\ y &\geq 4 \\ x, y &\in \mathbb{Z} \end{aligned}$$

[50] In particular, a good cut will always take advantage of the fact that some variables are forced to be integers.

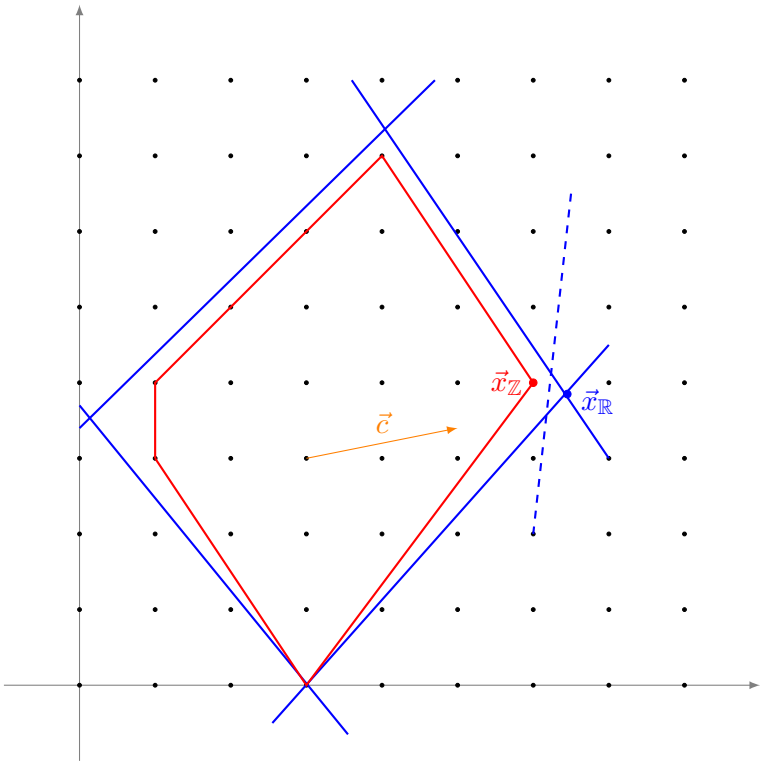

Figure 2: An integer program with objective function $\vec{c}$ and optimal solution $\vec{x}_{\mathbb{Z}}$.

Taking a positive linear combination of the first two, we get

$$\begin{array}{r} 2x + y \geq 1 \\ y \geq 4 \\ \hline 2x + 2y \geq 5 \end{array}$$

But for x and y integers, $2x + 2y$ must be an even integer, which means that we must have $2x + 2y \geq 6$. Furthermore, since the point $(x, y) = (-1.5, 4.5)$ satisfies the (relaxation of) the original constraints but not the new one, this is a good cut.

7.2.3.2 Disjunction cuts

Consider the constraints

$$\begin{aligned} -2x + 2y &\leq 1 \\ 2x + 4y &\leq 5 \\ x &\in \mathbb{Z} \end{aligned}$$

If these two inequalities form my basis, then the corresponding BFS is $(1/2, 1)$. One way to (artificially) enforce the constraint $x \in \mathbb{Z}$ would be to split into two different linear programs: one with the constraint $x \geq 1$ and the other with the constraint $x \leq 0$. A more clever method, however, would be to take each of these two possibilities and combine them with the given constraints in some way that provides a single inequality that must hold in both cases.

- If $x \geq 1$, then we can take the positive linear combination

$$\begin{array}{rl} 2x + 4y \leq & 5 \\ -3x \qquad \leq & -3 \\ \hline -x + 4y \leq & 2 \end{array}$$

- If $x \leq 0$, then we can take a different positive linear combination

$$\begin{array}{rl} 2x + 2y \leq & 1 \\ -\frac{3}{2}x \qquad \leq & 0 \\ \hline -\frac{1}{2}x + 2y \leq & 1 \end{array}$$

  which (after multiplying by 2) becomes $-x + 4y \leq 2$.

Hence in each case, $-x + 4y \leq 2$, so this is a valid inequality. Furthermore, since the point $(x, y) = (0.5, 1)$ satisfies the (relaxation of) the original constraints but not the new one, this is a good cut.

Notice that both incorporate something that is true for the original integer program P but which would not (necessarily) be true in the relaxation. There are other (usually number theoretic or combinatorial) ways to make cuts, but (to some extent) these are not necessary:

Theorem 23. *For any formulation P of a mixed integer linear program, there exists a series cuts (each of which is a rounding or disjunction cut) after which the new formulation $\widetilde{P}$ satisfies* $\text{feas}(\widetilde{P}) = \text{conv}(\text{feas}(\widetilde{P})) = \text{conv}(\text{feas}(P))$.

However we should expect that *any* method of producing cuts to either (1) be hard or (2) require an exponential[51] number of cuts (see Section 7.1.1).

7.2.4 References

1. Bertsimas and Tsitsiklis: Chapter 10, Section 11.1

7.3 Branch and Bound

Warning! Much of this section was covered in Week 8, but the topic is better suited for Week 7 (so I put it here).

Branch and Bound is one of the more common heuristic methods for solving[52] an integer program by breaking the problem into subproblems and (effectively) searching the entire space. The "branching" means to break a problem into two subproblems. The "bounding" The "bounding" means to keep the best solution seen so far and use this to eliminate subproblems. Typically, the branching is done by inserting a disjunction to cut the feasible region into two pieces (thus forming two subproblems). That is, you pick an integer combination of integer variables $\sum_i a_i x_i$ and some integer b and considering the two (new) integer programs that you get by adding one of the two (new) constraints:

$$\sum_i a_i x_i \leq b \quad \text{and} \quad \sum_i a_i x_i \geq b + 1.$$

[51] In the size of the original formulation.
[52] Trying to solve, of course.

How you pick these disjoints (and the order in which you solve the subproblems) is more art than science and is often the most important factor in determining whether you can effectively solve your problem (see Section 7.3.1).

As we divide the solution space further and further, we form a binary tree which is called the *Branch and Bound tree*. The root node corresponds to the original integer program and each child differs from its parent by exactly one constraint. Now imagine that we were able to solve the linear program relaxation at each node. Then parent-child relationship in the tree tells us the important fact:

> The value that we get for solving the LP relaxation at a parent will always be "at least as good" as the one we get when solving the LP relaxation at the child.

In particular, if a parent node LP relaxation is infeasible, the child node relaxation will also be infeasible. If our original problem was a maximization problem and the optimal value of a parent node LP relaxation is 17, then the optimal value of any LP relaxation which is lower on the tree will have optimal value at most 17. This allows us to "prune" the tree — once we get an infeasible LP relaxation, we can throw out all of the descendant nodes (they are infeasible). Similarly, if we have found a feasible solution to our original (maximization) integer program that has value 18, and then we find a node whose LP relaxation has optimal value 17, then we can throw out all of the descendant there as well. This is the "bounding."

In theory, a Branch and Bound tree could be infinite, but typically we are solving problems with a finite set of solutions, and then we can pick our disjunctions in such a way that at some point there is only one valid integer solution (disjunctions like $x \leq 0$ vs $x \geq 1$ will eventually assign a value to all of the binary variables, for example). Then the tree will have leaves and each of those leaves will correspond to a possible solution (though not necessarily a feasible one). In this case, a "brute force" solution to the problem could be found by simply testing all of the leaf nodes for feasibility and then taking the best one. Instead, we are proposing something much more complicated — to not only find a solution for *every node* (not just the leaves), but to solve a linear program at each node! So while this might seem like more work than is necessary, we can't judge it the way we judge a normal algorithm[53]. The problem with "checking every leaf" is that you always have to check every leaf, so that approach will always take the same amount of time (no matter what order you put the leaves in). If that amount of time is 15 years, does it really matter that (in a worst case scenario) Branch and Bound might take 20 years? The idea is to try something that might only take 30 minutes instead of doing something we know is never going to take 30 minutes.

I should say that the situation is not as dire as it sounds — yes, we are solving a different LP relaxation at each node, but if we have already solved the parent LP relaxation, then the child LP relaxation only differs by a single constraint. And while adding a single constraint can change the feasible region drastically, a constraint in the primal LP corresponds to a *variable* in the dual LP. And adding a variable is *much* simpler than adding a constraint[54] So there is a way to use the dual solutions of the parents to help solve the dual solutions of the children, at least in theory (no method for attacking an NP-complete problem here will "always work") and this turns out to be accurate in practice as well[55].

IN ANY CASE, solving the linear programs is the least of our concern — if we don't get enough prunings, this method will fail (it will eventually succeed, but not in a reasonable amount

[53] If you have not read Section 1.3 yet, now would be a good time.

[54] If you are at a BFS, a contraint can make you infeasible, so you have to go find another BFS. If you are at a BFS and then you add a variable, you can simply set that variable to 0 and you will be at a BFS again.

[55] One can find situations where a new constraint still makes the dual hard to solve, but these have to be constructed arefully, so in general one can hope that they don't appear in many natural settings.

of time). So everything comes down to picking a good branching and then also picking a good ordering to traverse the tree. Again, there is no "sure way" to do this, but if you understand the problem then you can often increase the odds significantly (see the next section). There are also clever modifications you can do like (for example) doing a "warm start" where you try to a few solutions that have a chance of being good to try to get a good baseline for your "bounding" and then starting at the top of the tree to try to cut as many branches away as you can. However, you can rest assured that NOTHING[56] you come up with will work 100% of the time, since (if it did), you would be able to use it to solve NP-complete problems (which we don't believe is possible).

There is one final "pruning" that can occur that is worth mentioning, and that is when you solve an LP relaxation and (magically) get an integer optimal solution. Is say "magically" but in practice this is something that happens more than one might think. Dependencies (see Section 7.2.1), for example, can often cause this to happen (forcing a single variable to a value can often cause other variables to be forced as well).

Example 24. As an example[57], we will consider a maximization MILP containing 4 binary variables. The disjunctions we will use are the simplest ones possible: for each binary variable t, we will consider the cases $t \leq 0$ and $t \geq 1$. This forms a tree with 16 leaves, which we will traverse using a heuristic called *iterative rounding*. The idea behind iterative rounding is that if we solve an LP relaxation and we find that the optimal solution has a variable $t = 0.8$, then we might suspect that the optimal integer solution is more likely to have $t = 1$ than $t = 0$.[58] We well let M be "the best value from a feasible solution we have seen so far" (so start with $M = -\infty$).

The following table gives the sequence of branch and bound steps: $\star$ means that we haven't branched on a given variable yet, so it will be allowed to take any value in the interval $[0, 1]$ when we do the LP relaxation. Figure 3 depicts the final search path of the Branch and Bound tree after all of the prunings have been made (in this example, we never actually reach a leaf node).

#	x	y	z	w	opt value	opt solution	action
1	$\star$	$\star$	$\star$	$\star$	25	$(0.1, 0.6, 0.7, 0.7)$	round x
2	0	$\star$	$\star$	$\star$	20	$(0, 0.2, 0.5, 0.7)$	round y
3	0	0	$\star$	$\star$	15	$(0, 0, 0.7, 0.4)$	round z
4	0	0	1	$\star$	$\varnothing$	infeasible	backtrack
5	0	0	0	$\star$	12	$(0, 0, 0, 0)$	set $M = 12$, backtrack
6	0	1	$\star$	$\star$	11	$(0, 1, 0.7, 0.2)$	cut branch, backtrack
7	1	$\star$	$\star$	$\star$	18	$(1, 0.2, 0.6, 0.3)$	round y
8	1	0	$\star$	$\star$	16	$(1, 0, 1, 0)$	set $M = 16$, backtrack
9	1	1	$\star$	$\star$	15	$(1, 1, 0.1, 0.4)$	cut branch, backtrack

The optimum value for this MILP is therefore $M = 16$ with solution $(x, y, z, w) = (1, 0, 1, 0)$.

Some comments:

- At each stage, we are solving an LP relaxation, and then using the solution to help us guess what a good guess for the next "branch" might be. In the worst case scenario, we might end up having to check the entire tree.

[56] Assuming P $\neq$ NP

[57] Note that there is a video that walks through this example: the Branch and Bound video starting at time 17:50.

[58] Note that even if the LP relaxation has an optimal solution with $t = 1$, that does *not* mean that will be the case in the original integer program. So this is *really* just a guess.

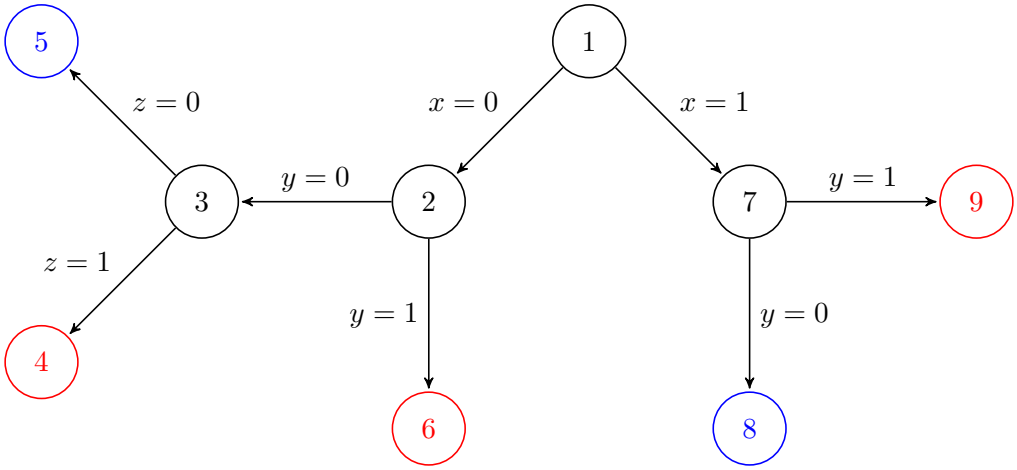

Figure 3: Example traversal of a Branch and Bound tree. Blue nodes returned solutions with integer values (so we can prune and update the max), and red nodes were places where the relaxation was infeasible or worse than the known max (so we can prune).

- The fact that steps (5) and (8) ended with an integer solution is the "magical" situation mentioned above. However, if this hadn't happened, you can see that the next step down would have been a leaf node, so we would have gotten an integer solution eventually (even if the magic didn't happen).
- We never needed to branch on w, which can often happen when you have dependent variables (like the ones we used in the modeling techniques from Section 7.2.1.
- If we had tried $x = 1, y = 1$ before $x = 1, y = 0$, we would not have been able to cut the branch (which could have cost us some extra steps). Again, either this was luck, or some underlying effect of our rounding strategy (but nothing we have control over).
- As we mentioned above, solutions to parent problems can often help you solve child problems (using the dual!) which is one reason why we rounded one variable at a time[59]

7.3.1 Personal experience

I have personal experience solving integer programs — the main technology that the company I worked for (Crisply) was trying to develop was a combination of machine learning and mixed integer linear programming as a way to do personalized classification, clustering, and estimation.

Imagine, for example, that you wanted to know how much time you spent studying for MATH-261. One way you could do this is to look at your computer history and see when your browser accessed our Moodle page. But how long did you stay there? And when you went to Facebook halfway through, did you spend half an hour there or just 30 seconds? If you accessed the Moodle page at 10:00 AM and then again at 11:00 AM, does that suggest you were working on MATH-261 for the full hour, or is it more likely that you took a break at 10:00 and then you started back again at 11:00?

Our goal was to try to take a collection of digital signals (like your access to webpages, files, etc) and try to predict what you were doing during those times. What we soon realized was

- two people with the same digital signals could have vastly different behaviors

[59] In theory, we could try to "round all the variables" right at the beginning, but then this quickly becomes more like the "brute force" method.

- if we showed the signals to someone who knew the person well (like their secretary or their parents or their roommates) those people could often tell you the correct way to interpret the signals[60]

So somehow we needed to have a decision making process that depended not only on available data, but also parameters associated to a person's personal style.

Our solution was to take the given data and find "the most reasonable guess of what you did, given your profile" by solving a huge MILP. A person's "profile" was incorporated into the **b** and **c** vectors (which in turn determined the constraints and objectives in the MILP). We would start with a generic profile, and then the users would correct our guesses and we would use the corrections to "learn" your style. And our primary method for solving our MILPs was using a modified version of Branch and Bound. One benefit of Branch and Bound was that the times when it did fail (that is, we decided it was taking too long), it was often still able to give us a "pretty good solution" since

- It is always keeping the "best solution it has seen so far" in memory, and this solution has (in theory) been compared to lots of different possibilities
- You can often use the LP relaxations solutions you did find to show that your "pretty good solution" is not far from optimal.

So if you are wondering whether this is really how people solve generic integer programs, the answer is definitely *YES*. I spent a non-trivial amount of my life working on the "art" of finding the best Branch and Bound path for a particular type of MILP and I can attest that understanding how this very specific collection of MILPs behaved was already quite complicated (forget trying to understand MILPs in general!). For certain problems, you can do more clever things[61] but at some point you will find a problem where you have to just start guessing at what a good solution should look like, and the LP relaxation (or some collection of LP relaxations) can be a very useful tool in trying to figure out the answer.

7.3.2 References

1. Bertsimas and Tsitsiklis: Section 11.2

[60] For example, your parents might know that you are the kind of person that tends to study in "blocks" so when they see you accessed Moodle at 10:00 and 11:00, they would say you probably were studying the whole time. Or maybe your roommate knows that you tend to get a snack at 10:45 everyday, so they would guess that you worked from 10:00 until 10:45, then got a snack, and then went back to working at 11:00.

[61] This is what we will discuss in much of the remaining part of class.

8 Week 8: Graphs and Networks

8.1 Graphs and Networks

An (undirected) graph $G = (V, E)$ consists of a ground set of *vertices*[62] V and a collection of (unordered) pairs of vertices called *edges*[63][64]. Given an edge $e = \{u, v\}$ we say that e is *incident* to u and v. The *degree* of a vertex v is the number of edges incident to that vertex.

A *walk of length* k in a graph G is a sequence of vertices $v_1, \ldots, v_k \in V$ such that $\{v_i, v_{i+1}\} \in E$ for all $i = 1, \ldots, k-1$. A *path* of length k is a walk for which all $v_1, \ldots, v_k$ are unique and a *cycle* of length k is a walk for which $v_1, \ldots, v_{k-1}$ are unique and $v_k = v_1$. A graph is called *connected* if there exists a path between any two vertices.

Example 25. The graph in Figure 4 has

$$V = \{1, 2, 3, 4, 5\} \quad \text{and} \quad E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 4\}, \{3, 4\}, \{3, 5\}\}.$$

Also, for example,

- $(1, 2, 3, 1, 4, 1, 3)$ is a walk
- $(4, 1, 2, 3)$ is a path
- $(1, 2, 3, 4, 1)$ is a cycle

Note that the definitions for walk, path, and cycle that we are using all depend on the ordering of the vertices (or *orientation*). So, for example, the cycles

$$C = (1, 2, 3, 4, 1) \quad \text{and} \quad C' = (1, 4, 3, 2, 1)$$

will be considered to be *different* cycles, even though they use the same vertices and edges.

The following lemma is the first one must people learn in graph theory:

Lemma 26 (Handshake)**.** *For all graphs* G

$$\sum_{v \in V} \deg(v) = 2|E|$$

A *tree* is a connected graph with no cycles. A vertex in a tree with degree 1 is known as a *leaf*.

Theorem 27. *The following are equivalent:*

- G *is a tree*
- G *is connected and has* $|V| - 1$ *edges*
- *there exists a unique path from* u *to* v *for all* $u, v \in V$*.*

Theorem 28. *Every tree on more than one node has at least* 1 *leaf.*[65]

A graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is called a *subgraph*. If $G = (V, E)$ is a connected graph, a subgraph of G with $|V|$ vertices that forms a tree is called a *spanning tree*. The blue edges in Figure 4, for example, form a spanning tree. A graph has a unique spanning tree if and only if it is a tree (typically one can find lots of possible spanning trees).

[62] Sometimes called *nodes*.
[63] Sometimes called *arcs*.
[64] Note that we do not allow multiple edges or self loops.
[65] They actually have at least 2 but 1 is easier to prove and is all we will need.

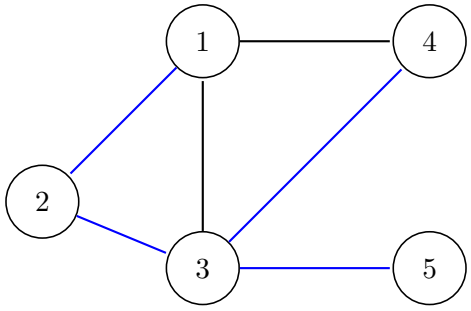

Figure 4: A graph. The blue edges form a spanning tree.

8.1.1 Digraphs

A directed graph (or digraph) $D = (V, E)$ is similar to a graph except that edges are *ordered pairs*. The (undirected) graph formed by turning (u, v) into $\{u, v\}$ (and removing duplicates) is called the *underlying graph*.

Given a (directed) edge $(u, v) \in E$, we will call v the *head* and u the *tail*. Edges for which v is a head are called *in-edges* at v and edges for which v is a tail are called *out-edges* at v. The *in-degree* of a vertex v is the number of in-edges at v and the *out-degree* is the number of out-edges at v.

An *undirected walk of length* k in a digraph D is a sequence of vertices and directed edges[66]

$$(v_1, e_{1,2}, v_2, e_{2,3}, \ldots, v_{k-1}, e_{k-1,k}, v_k)$$

for which $(v_1, \ldots, v_k)$ is a walk in the underlying graph and for which $e_{i,j} = (v_i, v_j)$ or $e_{i,j} = (v_j, v_i)$. Similarly for an *undirected path* and *undirected cycle*. Note that we have to designate vertices and edges when talking about undirected substructures of directed graphs. This is because a pair vertices $\{u, v\}$ could have two possible edges between them — (u, v) and (v, u) — and so one must designate which edge is being used. So, for example, the cycles

$$C = (2, (1, 2), 1, (1, 3), 3, (3, 2), 2) \quad \text{and} \quad C' = (2, (2, 1), 1, (1, 3), 3, (3, 2), 2)$$

will be considered to be distinct cycles, even though they have the same vertices and same orientations.

A *directed walk* in a digraph D is a sequence of vertices $v_1, \ldots, v_k \in V$ such that $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, k - 1$. Similarly for a *directed path* and *directed cycle*.[67]

Example 29. The graph in Figure 5 has

$$V = \{1, 2, 3, 4, 5\} \quad \text{and} \quad E = \{(1, 2), (2, 1), (3, 2), (4, 3), (1, 4), (1, 3), (3, 5)\}$$

and its underlying graph is the graph from Figure 4[68] Also, for example,

- $(2, (1, 2), 1, (1, 4), 4)$ is an undirected path
- $(4, 3, 2, 1)$ is a directed path

[66] Because there are can now be two edges between any two vertices, we must specify which one is used.

[67] If we know that a given path/walk/cycle is a directed path/cycle/walk then we don't need to include the edges because there is only one edge that goes in the correct direction.

[68] Note that the underlying graph, in this case, has 1 fewer edge than the original graph since the two edges between 1 and 2 become a single edge.

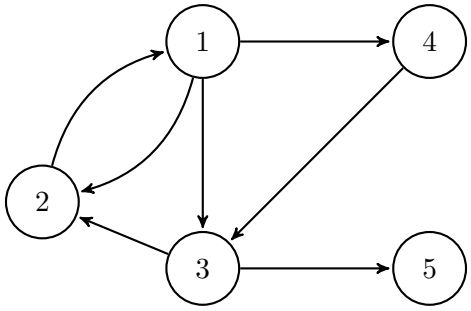

Figure 5: A digraph.

- $(2, (1,2), 1, (1,3), 3, (3,2), 2)$ is an undirected cycle
- $(1, 3, 2, 1)$ is a directed cycle

Note that, just as in the case of undirected graphs, our definitions for undirected walk, path, and cycle will again depend on orientation. So, for example, the cycles

$$C = (2, (1,2), 1, (1,3), 3, (3,2), 2) \quad \text{and} \quad C' = (2, (3,2), 3, (1,3), 1, (1,2), 2)$$

will again be considered to be distinct cycles even though they use the same vertices and edges.

8.1.2 Network flows

A *network* is a directed graph with weights on the vertices and nonnegative weights on the edges. The weight given to a vertex is called the *supply* at that vertex. A vertex with positive supply is known as a *source* and a vertex with negative supply is known as a *sink*[69]. The nonnegative weight on an edge is called the *capacity* of that edge.

Let $D = (V, E)$ be a digraph with vertex weights $b(v)$ and nonnegative edge weights $C(e)$. A function $f : E \to \mathbb{R}$ is called a *flow*. We will often think of a flow as a transfer of goods from one vertex to another. The supply of a vertex will then be thought of as the amount of goods that a given vertex produces (if it is a source) or consumes (if it is a sink) and the capacity of an edge will be the maximum amount of goods that can be sent from one vertex to another along that edge. A flow is called *feasible* if

- $0 \leq f(e) \leq C(e)$ for all $e \in E$.
- For all $v \in V$
$$b(v) + \sum_{e \in I(v)} f(e) - \sum_{e \in O(v)} f(e) = 0 \tag{19}$$
where $I(v)$ denotes the set of in-edges at v and $O(v)$ the set of out-edges at v.

Equation 19 is known as the *flow conservation law* — it says that there can be no excess or shortage of goods at a given vertex. Notice that each edge has a head and a tail. Hence

$$\sum_{v \in V} \left(\sum_{e \in I(v)} f(e) - \sum_{e \in O(v)} f(e) \right) = 0$$

[69] A vertex with 0 supply doesn't really have a name — we just say that it is non-source, non-sink vertex.

Hence if we sum (19) for all $v \in V$, we get

$$\sum_{v \in V} b(v) = 0$$

which tells us that any feasible flow will leave the system in an equilibrium (the net amount of goods is 0).

The most general network flow problem contains a *cost* function $c(e)$ which is the cost for using a given edge. The goal is then to find a feasible flow (if one exists) that minimizes the total cost: $\sum_e f(e)c(e)$. Note that if the goods being transported can be delivered in fractional amounts (like when the flow is water), then this is a linear program. Otherwise (like when the flow is people) it is an integer program. However, to put it into our usual "vector/matrix" form, we will find it useful to treat V and E like vector spaces.[70]

The *incidence matrix* of a digraph $D = (V, E)$ is a $V \times E$ matrix $\mathbf{A}$ with entries

$$A_{v,e} = \begin{cases} +1 & \text{if } v \text{ is the tail of } e \\ -1 & \text{if } v \text{ is the head of } e \\ 0 & \text{otherwise.} \end{cases}$$

If D a digraph with incidence matrix $\mathbf{A}_D$ and D' is a sub-digraph of D, then the incidence matrix of D' is the submatrix of $\mathbf{A}_D$ corresponding to those vertices and edges remaining in D'.

Example 30. Let D be the digraph in Figure 5 with V and E ordered as in Example 29. Then the incidence matrix is

$$\mathbf{A}_D = \begin{bmatrix} + & - & 0 & 0 & + & + & 0 \\ - & + & - & 0 & 0 & 0 & 0 \\ 0 & 0 & + & - & 0 & - & + \\ 0 & 0 & 0 & + & - & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & - \end{bmatrix}$$

where we are using $+$ to denote $+1$ and $-$ to denote -1 to (hopefully) make it easier to read.

We can then rewrite (19) as

$$\mathbf{b} = \mathbf{A}\mathbf{f} \tag{20}$$

where $\mathbf{f} \in \mathbb{R}^E$ is called the *flow vector* and $\mathbf{b} \in \mathbb{R}^V$ is called the *supply vector*.

Note that incidence matrices are never full rank (since $\mathbf{1}^T\mathbf{A} = \mathbf{0}$). However, the next theorem shows that they are almost full rank (when the underlying graph is connected), as you will prove in the problem set.

Theorem 31. *Let $D = (V, E)$ be a digraph with incidence matrix $\mathbf{A}$ whose underlying graph is connected. Then* $\operatorname{rank}(\mathbf{A}) = |V| - 1$.

In fact, we will see that every set of $|V| - 1$ edges in D that forms a spanning tree in the underlying graph will correspond to a rank $(|V| - 1)$ submatrix of $\mathbf{A}$.

[70] This means we will index the coordinates of our vectors with vertices (or edges) rather than integers.

8.1.3 Circulations

A flow $\mathbf{f}$ that is in the kernel of an incidence matrix $\mathbf{A}$ is called a *circulation*. That is, if

$$\mathbf{Af} = \mathbf{0}.$$

It is easy to see from (20) that if $\mathbf{f}_1$ and $\mathbf{f}_2$ are feasible flow vectors, then their difference $\mathbf{f} = \mathbf{f}_1 - \mathbf{f}_2$ is a circulation. We will see that a particularly useful set of circulations in a digraph D are the ones that correspond to undirected cycles in D. Given an undirected cycle

$$C = (v_1, e_{1,2}, v_2, \ldots, v_{k-1}, e_{k-1,1}, v_1)$$

we associate to it the *cycle vector* $\mathbf{h}^C$ with

$$h_e^C = \begin{cases} +1 & \text{if } e = (v_i, v_{i+1}) \\ -1 & \text{if } e = (v_{i+1}, v_i) \\ 0 & \text{otherwise.} \end{cases} \tag{21}$$

Note that this definition depends not only on the edges that are in a cycle, but also the order in which they are listed. The cycle C' which uses the same edges as C but goes in the opposite direction will have $\mathbf{h}^{C'} = -\mathbf{h}^C$. The edges which get assigned a $+1$ are called *forward* edges[71] and those that get a -1 are called *backward* edges.

Example 32. Given the ordering of the edges in Example 29, the cycle

$$C = (1, (1,2), 2, (3,2), 3, (4,3), 4, (1,4), 1)$$

will have cycle vector $\mathbf{h}^C = (1, 0, -1, -1, -1, 0, 0)^\intercal$ — see Figure 6. Hence $(1,2)$ is a forward edge and $(3,2)$ is backward in C (and they would be reversed in C'). One can check using the matrix in Example 30 that

$$\mathbf{A}_D\mathbf{h}^C = \begin{bmatrix} + & - & 0 & 0 & + & + & 0 \\ - & + & - & 0 & 0 & 0 & 0 \\ 0 & 0 & + & - & 0 & - & + \\ 0 & 0 & 0 & + & - & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & - \end{bmatrix} \begin{bmatrix} + \\ 0 \\ - \\ - \\ - \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

so $\mathbf{h}^C$ forms a circulation (as do all cycle vectors).

8.1.4 Cuts

Let s and t be two distinct vertices in a digraph. We define an $s-t$ *cut* in D to be a subset of vertices $S \subset V$ for which $s \in S$ and $t \notin S$. Edges (u, v) where $u \in S$ and $v \notin S$ are said to *cross* the cut. The *capacity* of an $s-t$ cut is the sum of the capacities of all edges that cross the cut going from S to $V \setminus S$. That is

$$C(S) = \sum_{v_i \in S, v_j \notin S} C(v_i, v_j)$$

where we assume $C(e) = 0$ for any $e \notin E$.

[71] Because if you actually walked the cycle as it is described, the direction you would need to walk on that edge is the same as the direction it is pointing.

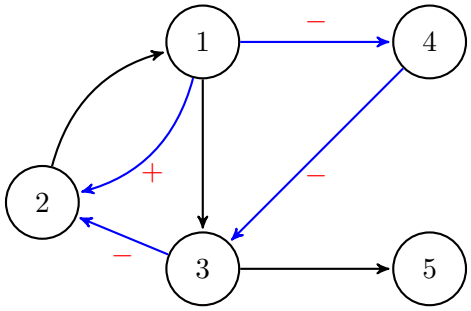

Figure 6: The cycle C (in blue) from Example 32 and the nonzero entries of $\mathbf{h}^C$ (in red).

Example 33. Let $D = (V, E)$ be the graph in Figure 7. The two $s - t$ cuts (with their capacities) are:

- the red cut (with capacity 3), and
- the green cut (with capacity 8).

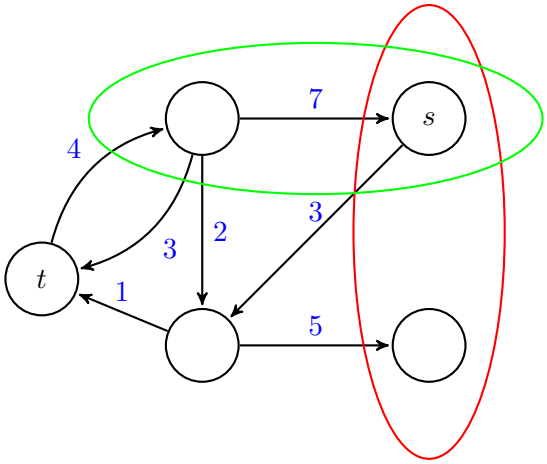

Figure 7: A network with edge capacities (in blue) and two different $s - t$ cuts.

Just like with cycles, there is a natural vector associated to an $s - t$ cut K. The *cut vector* of K is the vector $\mathbf{r}^K$ with

$$r^K_{(u,v)} = \begin{cases} +1 & \text{if } u \in S, v \notin S \\ -1 & \text{if } v \in S, u \notin S \\ 0 & \text{otherwise} \end{cases}. \tag{22}$$

Hence $\mathbf{r}^K$ records which edges belong to a cut, as well as the direction they are going in — see Figure 8 for an example. Similar to the case of cycles, we will refer to the edges which get assigned a $+1$ *forward* edges and those that get a -1 *backward* edges.

Example 34. Given the ordering of the edges in Example 29, the cut

$$K = \{2, 4\}$$

will have cut vector $\mathbf{r}^K = (-1, +1, -1, +1, -1, 0, 0)^\intercal$ — see Figure 8.

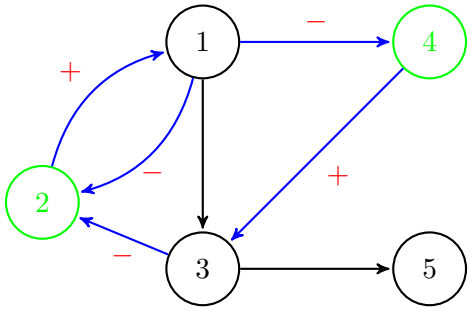

Figure 8: The cut $K = \{2, 4\}$. Edges crossing K are blue and nonzero values of $\mathbf{r}^K$ are red.

Note that $\mathbf{r}^K$ (from Example 34) and $\mathbf{h}^C$ (from Example 32) have the interesting property that

$$\mathbf{h}^C \cdot \mathbf{r}^K = (+, 0, -, -, -, 0, 0)^\intercal \cdot (-, +, -, +, -, 0, 0)^\intercal = 0.$$

A fundamental theorem (which we will prove on the problem set) says that this is always the case

Theorem 35. *Every cut vector* $\mathbf{r}^K$ *and every cycle vector* $\mathbf{h}^C$ *satisfy*

$$\mathbf{h}^C \cdot \mathbf{r}^K = 0$$

It is worth noting that Theorem 35 is not obvious. When we first started writing the flow weights as a vector, it seemed like we were just doing it for notational reasons (so that we could use $\mathbf{b} = \mathbf{A}_D\mathbf{f}$ to write the flow feasibility constraints, for example). I could take any list of numbers (like the average temperatures in a year or my bowling scores, for example) and write it as a vector, but that certainly doesn't mean that I should expect "the space of vectors that are orthogonal to the temperatures in the year 2000" to have any meaning. So the fact that two natural structures that exist in a graph (cuts and cycles) also happen to have linear algebra relationships suggests that something much deeper is happening here (and we will soon see that this is correct).

8.1.5 Orientation

I often find that some people get confused regarding the difference between "directedness" and "orientation." Directedness has to do with which way the edges are pointed. Orientation has to do with which way the vertices/edges are ordered in your description of that object. Orientation will be a factor for many of the objects we discussed — cycles, paths, walks, and even cuts (if we don't say which side s and t are in, then the there is no way to distinguish the cuts that have cut vectors $\mathbf{r}^K$ and $\mathbf{r}^{-K}$ (for example). The general rule (for this class) is:

> Two things (cycles/paths/walks/cuts) that consist of the same vertices and edges but have different orientations are to be treated as two *different* objects.

So (for example) the path $1 \to 2 \to 3$ is different than the path $3 \to 2 \to 1$ (even though it uses the same vertices and edges). Unfortunately, I can *not* guarantee that this will be the case elsewhere in life, but it is important to know that this issue exists so you can ensure that you know what the correct definition is wherever you find it.

In any case, a few comments regarding cuts and cycles:

- Cuts

  Cuts can be a bit confusing because (as mentioned before) there exists a concept of a "cut" in graph theory which is not considered to have a "orientation" because there is no way to tell the difference between the two sides — the cut defined by a set S would be the same as the cut defined by a set $V \setminus S$. However in this class, we will not use this concept — in this class, "cut" always means "$s - t$ cut" which means that is oriented. The side with s in it is always the "starting" side and the side with the t is always the "finishing" side. A better name would actually be $s \to t$ cut, but I want to match the definitions in the book (even if they are not great). See also the part on Capacities below.

- Cycles

  Cycles can be confusing for the same reason as cuts — often you want to look at a collection of edges and say "That is a cycle." However in this class, we will need to say more — not just the edges involved but also the order in which we are considering the edges to appear in (this is completely independent of which *direction* the edges are pointing). However, this is not the worst of the matter: historically, people tend to simply use the word "cycle" in a directed graph without signifying whether the qualifications for being a cycle are that the edges all point the same way[72]. In particular, in the remaining sections in the course, the concept of a "negative cycle" will appear in two contexts — one where it means "undirected cycle" and the other where it means "directed cycle." The first will be when we discuss the "negative cost cycle algorithm" and we will look for *undirected cycles* that have negative cost (so cycles in the underlying graph). However when we discuss the `SHORTEST-PATH` problem, we will see the term "negative length cycle" again which in that case refers to a *directed cycle*[73].

- Cut vectors and cycle vectors

  Again, some people might consider cut vectors and/or cycle vectors to be unoriented (because they consider cuts and/or cycles to be unoriented as in the last item). However we will always consider them to be oriented, which should be clear from their definitions in (21) and (22).

- Capacities

  Capacities (in this class) will only be applied to directed graph and are always (themselves) oriented (meaning we only want the values of edges going in a certain direction). If at some point in your life you meet someone who wants to assign a "capacity" to an edge in an undirected graph, then you should turn it into a directed graph by splitting each edge $e = \{u, v\}$ into two new edges

  $$e_1 = (u, v) \quad \text{and} \quad e_2 = (v, u)$$

  and then set $C(e_1) = C(e_2) = C(e)$. Similarly with cuts — the capacity of a cut in this class always means the edges going from the s part to the t part.

8.1.6 References

1. Bertsimas and Tsitsiklis: Sections 7.1–7.2

[72] And I am sure I will make this mistake at some point — if there are ever questions, please ask!

[73] Actually, all of the terminology regarding so-called "shortest path" problems is pretty bad, so maybe this should not come as a surprise.

9 Week 9: Network Flows meet Simplex!

Let $D = (V, E)$ be a digraph with incidence matrix $\mathbf{A}$. Given a cost vector $\mathbf{c}$, a supply vector $\mathbf{b}$ and a (nonnegative) capacity vector $\mathbf{C}$, the general network flow problem is

$$\begin{aligned} \min\ & \mathbf{c} \cdot \mathbf{f} && (23)\\ \text{s.t. } & \mathbf{A}\mathbf{f} = \mathbf{b} && (24)\\ & \mathbf{f} \leq \mathbf{C} && (25)\\ & \mathbf{f} \geq \mathbf{0} && (26)\\ & \mathbf{f} \in \mathbb{Z}^{|E|} \quad (*) && (27) \end{aligned}$$

where the $(*)$ constraint is optional (depending on whether the solution is required to be integral or not). For the moment, we will get rid of $(*)$ and come back to it later.

The special case when all capacities are infinite is called the *uncapacitated* network flow problem:

$$\begin{aligned} \mathcal{P} = \min\ & \mathbf{c} \cdot \mathbf{f}\\ \text{s.t. } & \mathbf{A}\mathbf{f} = \mathbf{b}\\ & \mathbf{f} \geq \mathbf{0} \end{aligned}$$

and this should be quite familiar to us (because it is a linear program in equality standard form). We have already seen the interpretation of this as a problem on the directed graph D and the goal of this lecture will to understanding what the graph theory interpretations of the simplex method are.

9.1 Network simplex

In what follows, we will assume that the underlying graph of D is connected. If this is not the case, then we can break the problem up into a collection of problems, one for each connected component. Since these components do not effect each other in any way, we can consider them separately (which means each of the subproblems will be be connected).

9.1.1 The dual

Before getting into simplex, however, let us do what all good Discrete Optimization students should do when they see a brand new linear program they want to understand: form the dual (just so we can see what it looks like).

$$\begin{aligned} \mathcal{D} = \max\ & \mathbf{b} \cdot \boldsymbol{\lambda}\\ \text{s.t. } & \boldsymbol{\lambda}^\intercal \mathbf{A} \leq \mathbf{c}^\intercal \end{aligned}$$

This is some other "graph theory problem" but what exactly? Some observations:

- $\boldsymbol{\lambda}$ assigns a value to each *vertex* (as opposed to $\mathbf{f}$ which assigned a value to each edge).
- Each column of $\mathbf{A}$ consists of only 2 nonzero entries — a -1 at the head of the edge and $+1$ at the tail.

Hence in graph theory terms, we can write this as

$$\begin{aligned} \max \ & \sum_v \lambda(v) b(v) \\ \text{s.t. } & \lambda(u) - \lambda(v) \leq c(u,v) \qquad \text{for all } (u,v) \in E \end{aligned}$$

This prompts the following definition: given a vector $\boldsymbol{\lambda} \in \mathbb{R}^V$, we will call the flow defined by $\boldsymbol{\lambda}^\intercal \mathbf{A}$ the *induced flow*. Hence the dual problem is asking for the optimal vertex weighting for which the induced flow is bounded above by $\mathbf{c}$.

We will return to this in a later lecture, but it is worth noting that we have already seen a very important collection of induced flows: the ones that come from cuts. In fact, if $K \subset V$ is a cut, then the induced flow defined by

$$\lambda(v) = \begin{cases} 1 & \text{if } v \in K \\ 0 & \text{if } v \notin K \end{cases}$$

is precisely the cut vector $\mathbf{r}^K$ defined in (22).

9.1.2 Tree solutions

Now we return to solving the original problem $\mathcal{P}$ via the simplex method where (as noted above) we are assuming the underlying graph of D is connected. Our first obstacle is that (as we saw in the previous lecture) $\mathbf{A}$ is not full rank. In particular $\mathbf{1}^\intercal \mathbf{A} = \mathbf{0}$ which means that we need $\mathbf{1} \cdot \mathbf{b} = 0$ or else $\mathcal{P}$ is infeasible. On the other hand, if we are in the case that $\mathbf{1} \cdot \mathbf{b} = 0$ then one of the constraints $\text{row}_k(\mathbf{A}) = b_k$ is not really telling us much information (since it is a linear combination of the other constraints). So we will not lose anything by throwing it[74] away.

The $(|V|-1) \times |E|$ matrix formed by removing a row of $\mathbf{A}$ is called the *truncated incidence matrix* and we will denote it $\tilde{\mathbf{A}}$ (removing the same row from $\mathbf{b}$ gives the vector $\tilde{\mathbf{b}}$). A corollary of Theorem 31 is that

Corollary 36. *If D is a digraph whose underlying graph is connected, then its truncated incidence matrix $\tilde{\mathbf{A}}$ has full (row) rank.*

So if we assume that the original problem $\mathcal{P}$ has a feasible solution, then we will get the same solution by solving

$$\begin{aligned} \tilde{\mathcal{P}} = \min \ & \mathbf{c} \cdot \mathbf{f} \\ \text{s.t. } & \tilde{\mathbf{A}}\mathbf{f} = \tilde{\mathbf{b}} \\ & \mathbf{f} \geq \mathbf{0} \end{aligned}$$

The proof of Theorem 31 uses the fact that the columns of $\mathbf{A}$ corresponding to the edges of a spanning tree are linearly independent. Hence we define a flow vector $\mathbf{f}$ to be a *tree solution* if there exists a set of $|V|-1$ edges T such that

1. the edges from T form a spanning tree in the underlying graph.
2. $f_e = 0$ whenever $e \notin T$.
3. $\tilde{\mathbf{A}}\mathbf{f} = \tilde{\mathbf{b}}$.

[74] The convention is to remove the last constraint, but "last" really depends on the order you put the vertices in, so any constraint would be fine to remove.

If $\mathbf{f} \geq \mathbf{0}$, then we call it a *feasible tree solution.* You will prove the following theorem as part of your problem set:

Theorem 37. *A flow vector is a basic solution to $\tilde{\mathcal{P}}$ if and only if it is a tree solution.*

Two comments:

- Multiple trees can correspond to the same tree solution (corresponding to a degeneracy in the linear program).
- Since tree solutions are basic solutions (they can be computed using the basis matrix $\mathbf{B}$ (which is the $(|V|-1) \times (|V|-1)$ submatrix of $\tilde{\mathbf{A}}$ corresponding to the edges in the tree).
- The $\mathbf{B}$ associated to a tree solution is quite nice in two respects[75]. :
  - Find $\mathbf{f}$ by our usual way (calculating $\mathbf{B}\mathbf{f}_\beta = \mathbf{b}$ and setting all other entries of $\mathbf{f}$ to 0) can be done very efficiently (in particular, without needing to invert $\mathbf{B}$).
  - $\det[\mathbf{B}] = \pm 1$ (the relevance of this will become clear later)

9.1.3 Change of basis

We have seen that a basic solution corresponds to a spanning tree T in D and so adding a column to the basis is like adding an edge e to the tree (which will create a cycle). Since we know that basic solutions are trees, we can conclude that simplex will pick one of the edges in this new cycle, and remove it (forming a new tree). See Figure 9.

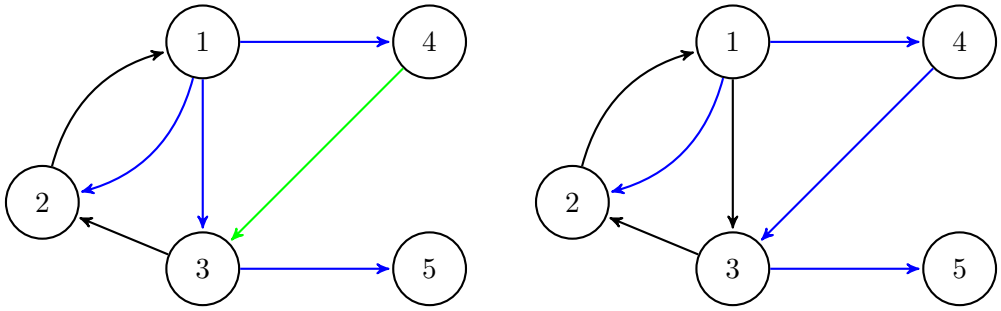

Figure 9: A tree solution (in blue). Simplex adds the green edge and then removes an edge (from the newly created cycle) to get back to a tree.

Assume that we added edge e. This forms two (undirected) cycles (see Section 8.1.5) with opposite orientations. Consider the one where e is pointing forward. If there are no degeneracies, then the act of putting e into the basis corresponds to moving $f_e = 0$ to $f_e = \theta$ for some θ. To accomplish this (while still remaining feasible), we can consider new solutions of the type

$$\mathbf{f} \to \mathbf{f} + \theta \mathbf{h}^C$$

where $\mathbf{h}^C$ is the cycle vector for C. The reduced cost associated to $\mathbf{h}^C$ will then be $\theta(\mathbf{c} \cdot \mathbf{h}^C)$ and so we will only consider those C for which this reduced cost is negative. Note that

$$(\mathbf{f} + \theta \mathbf{h}^C)_t = \begin{cases} f_t & \text{if } t \notin C \\ f_t + \theta & \text{if } t \in C \text{ is pointing forward} \\ f_t - \theta & \text{if } t \in C \text{ is pointing backwards} \end{cases}$$

[75] We will prove these on the problem set.

so the edge that would be removed is the backward-pointing edge that currently has the smallest flow value.[76]

However, we can use our knowledge of simplex to compute the reduced costs in a different way — using the dual solution. When $\mathbf{B}$ is a matrix associated to a tree, it is easy to solve $\mathbf{c}_\beta^\intercal = \boldsymbol{\lambda}^\intercal \mathbf{B}$ and so the reduced costs can then be easily computed:

$$\overline{\mathbf{c}} = \mathbf{c} - \boldsymbol{\lambda}^\intercal \tilde{\mathbf{A}}$$

so that

$$\overline{c}_{(i,j)} = c_{(i,j)} - \lambda_i + \lambda_j.$$

9.2 A "graph theory" algorithm

Now that we have seen what simplex is doing to solve this problem, let's try to describe it in the language of graph theory:

1. Start with a tree solution T.
2. For each edge e not in T, try adding it to T. This will result in a cycle C (oriented so e points forward).
   - If $\mathbf{h}^C \cdot \mathbf{c} \geq 0$ for all cycles formed this way, you are at an optimal solution.
   - Otherwise, choose a cycle C with $\mathbf{h}^C \cdot \mathbf{c} < 0$.
3. Look at the directions of the edges in C.
   - If every edge of C points forward, the solution is unbounded.
   - Otherwise, let e' be the smallest valued backward-pointing edge
4. Remove e' from T and insert e, forming a new tree T'.
5. Iterate.

Some observations:

- It still might be hard to find an initial BFS.
- This entire process is "closed under ideals". What I mean by this is that if you have a tree solution where all the values are multiples of t, then *every* tree solution will have values that are a multiple of t.

The second statement is notable because it means that if you have any *integer* tree solution, then every basic feasible solution will be an integer solution (so any solution to the LP relaxation will be a feasible solution to the original problem). As it turns out, this can then be related back to $\mathbf{b}$ and $\mathbf{c}$ by way of the basis matrix. The following theorem will be proved in the problem set.

Theorem 38. *Let $\mathbf{B}$ be a basis matrix corresponding to a tree solution. Then $\mathbf{B}^{-1}$ has only integer entries.*

Hence if $\mathbf{b}$ contains only integers, then every primal basic solution will contain only integers and if $\mathbf{c}$ contains only integers, then every dual basic solution contains only integers! So now we can return to the optional $(*)$ constraint and see that it is completely irrelevant. If we form $\mathcal{P}^*$ by adding the constraint $\mathbf{f} \in \mathbb{Z}^E$ to $\mathcal{P}$, then there are two possibilities:

[76] If every edge points forward and $\mathbf{c} \cdot \mathbf{h}^C < 0$, then the solution is unbounded.

- If $\mathbf{b}$ has a non-integer value, then $\mathcal{P}^*$ is infeasible[77].
- If $\mathbf{b}$ has only integer values, then the optimal solution for $\mathcal{P}$ found by simplex[78] will be optimal for $\mathcal{P}^*$ as well.

9.2.1 Capacitated problems

Finally, let's return to the more general network flow problem (where capacities are allowed). One way we could proceed is to (again) look at how simplex behaves, but this will be a little more complicated because we will need to first put things into equality standard form. So instead, let's take the graph theory perspective we just made and see if we can adapt it directly. We will still be focused on tree solutions, however now there will be two ways that an edge can be "not in the tree": they can be 0 or they can have full capacity. When an edge e enters the basis with $f_e = 0$, then the flow adds (as above) but if it enters with $f_e = C_e$, then the flow subtracts[79]. Similarly, all edges in the cycle will have a chance to exit the basis (regardless of what direction they point), since those for which the flow is subtracting could potentially hit 0 and those for which the flow is adding could potentially hit capacity. The exiting edge will be the one that achieves one of these two events first. Everything else remains the same.

9.2.2 References

1. Bertsimas and Tsitsiklis: Section 7.3
2. Video lecture on Capacitated Network Flow using gadgets

9.3 Negative Cost Cycle Algorithm

So now that we know what simplex is doing, is there any way to improve on it? At each iteration, simplex looks for a cycle that has negative cost, and then it adjusts the flow on that cycle to get a better solution. However simplex only tries to look at a small number of candidates — if we are at a basis β corresponding to tree T, the only cycles that simplex looks at (in that iteration) are the ones it can form by adding a single edge to T.

So one can ask the following question: what if I just looked for arbitrary negative cost cycles that I could add to my current solution? This approach leads to an algorithm known as the *Negative Cost Cycle algorithm (NCCA)*. To describe the algorithm in full generality, we will consider the network flow problem

$$\begin{aligned} \min\ & \mathbf{c} \cdot \mathbf{f} \\ \text{s.t. } & \mathbf{A}\mathbf{f} = \mathbf{b} \\ & \mathbf{f} \leq \mathbf{u} \\ & \mathbf{f} \geq \boldsymbol{\ell} \end{aligned}$$

where $\boldsymbol{\ell}$ and $\mathbf{u}$ are lower and upper bounds for the allowed flow (respectively). We will say a cycle C is *saturated* if one of the following occurs:

- There exists an edge e that is pointing forward in C and has $f_e = u_e$

[77] If $\mathbf{f} \in \mathbb{Z}^E$ then $\mathbf{A}\mathbf{f}$ has only integer values (and so there is no way to have $\mathbf{A}\mathbf{f} = \mathbf{b}$).

[78] Note that we need the phrase "found by simplex" in this statement for it to be true — it is possible that there are some (non-basic) optimal solutions in $\mathcal{P}$ that are not integral and so are infeasible in $\mathcal{P}^*$.

[79] In particular, for each edge e, we need to consider both cycles that can be formed by adding e — the one where e is pointing forward *and* the one where e is pointing backwards.

- There exists an edge e that is pointing backward in C and has $f_e = \ell_e$

Otherwise we call C an *unsaturated* cycle[80]

The NCCA is the following greedy procedure:

1. Start with a feasible flow $\mathbf{f}$[81]
2. Choose* an unsaturated cycle[82] C with negative cost ($\mathbf{c} \cdot \mathbf{h}^C < 0$).
   - If no such cycle exists, terminate and declare the current $\mathbf{f}$ optimal.
   - If $\mathbf{f} + \theta \mathbf{h}^C$ remains feasible for all $\theta > 0$, terminate and declare the solution to be unbounded.
   - Set $\mathbf{f} \leftarrow \mathbf{f} + \theta \mathbf{h}^C$ where θ is as large as possible
3. Iterate.

Some comments:

- This is similar in spirit to what simplex does (picking an unsaturated cycle with negative cost and adding it), but simplex only looks for this cycle locally in the polytope. The negative cost cycle algorithm will typically travel through the interior of the polytope (which can be both a good thing and a bad one).
- If the algorithm terminates, then it is correct.[83]
- There is no guarantee that the algorithm will ever terminate. Like simplex, there is always some way to choose* that will end up finishing, but there is no guarantee that *every* way will terminate. Hence one would still need to find a "choosing rule" like Bland's rule to guarantee termination.[84]

The third point here (non-termination) is an interesting one because it is for a completely different reason as simplex. Recall simplex had the issue that it might have degeneracies, which would cause the cost to improve by 0. This could lead to just going back and forth between bases living inside a single degeneracy. Assuming this didn't happen, we knew that simplex would converge because we would decrease the cost each time and there were only a finite number of available bases to pick from. On the other hand, the NCCA will always decrease the cost at every step, but the fact that we are moving through the interior of the polytope means we can no longer guarantee that the number of possible feasible solutions we can get is finite. For example, one could get a sequence of costs like $(2, 1.5, 1.25, \dots)$ which keeps getting closer to 1 but never reaches 1 (and this *can* happen, as we will see in the problem set). However, when we are in the case of integer flows, such a sequence can never happen (because you can't get infinitely close to something using decreasing integers).

9.3.1 References

1. Bertsimas and Tsitsiklis: Section 7.4 but don't worry too much about understanding the specifics of the NCCA. Its importance comes from the fact that (a) it is an example of how one can make a new algorithm from seeing how simplex solves a problem and (b) we will use it as a way to motivate the augmenting path algorithm in the next section.

[80] Note that the definitions of "saturated" and "unsaturated" only make sense if the cycle has an orientation.

[81] Assuming we can find one, and this may need simplex!

[82] Just to remind you, we mean *oriented* cycle here — so that means vectors $\mathbf{h}^C$ and $-\mathbf{h}^C$ are different cycles!

[83] The proof of this is a bit technical (it uses a flow decomposition theorem that I don't want to spend time proving) so I am going to avoid it here.

[84] Which can be done, but again is outside the scope of the lecture.

10 Week 10: Max Flow / Min Cut

10.1 MAX-FLOW

The first problem in the latest Problem Set gave a preview to this lecture, where we will look at a slight variation of the general network flow problem known as MAX-FLOW. In this variation, we are given a digraph $D = (V, E)$ with capacities on the edges $\mathbf{C} \geq \mathbf{0}$, but there is no supply vector $\mathbf{b}$. Instead, we are given two vertices s and t where s is the "global source" and t is the "global sink." All other vertices are neutral (do not make or consume any product), and the idea is for s to ship as many of the items it makes (we'll say "clams") to t as it can, while satisfying the capacity constraints and the flow conservations laws. The reason this is not a standard "general network flow problem" like we looked at in Section 8.1.2 is that the thing that MAX-FLOW is trying to optimize is part of the supply vector $\mathbf{b}$, rather than a cost. Nonetheless, we can turn this into a standard "general network flow problem" by a clever transformation, but first we should make some comments:

- Every instance of MAX-FLOW will have at least one feasible solution, because $\mathbf{0}$ satisfies the capacity constraints and the flow conservation laws (and gives a solution value of 0, because nothing is being moved).
- We can always find an optimal solution $\mathbf{f}$ of MAX-FLOW for which the nonzero edges form a subgraph with no directed cycles. A directed cycle would simply be moving clams in a circle, so removing those clams from the flow would give an answer with the same number of clams getting from s to t.
- Because of the previous comment, we can always find an optimal solution where there is no flow on edges that point to s and no flow on edges pointing away from t. Therefore, we can simplify the problem (without changing the optimal value) by simply getting rid of any such edges.

So now that we can assume that our digraph D has no edges pointing towards s or away from t, the first thing we will do is make a new digraph by adding a single new edge that points toward s and away from t. We will call that new edge $\underline{e}$ and the new digraph that we get from adding it $\underline{D}$. Furthermore, $\underline{e}$ will have no capacity (or ∞ capacity, if you prefer) and we are going to give it a cost of -1. All other edges will keep their previous capacities and be given a cost of 0. See Figure 10.

The claim is that the network flow problem

$$\begin{aligned} \mathcal{P} = \min \quad & -f_{\underline{e}} \\ \text{s.t.} \quad & \underline{\mathbf{A}}\mathbf{f} = \mathbf{0} \\ & \mathbf{f} \leq \mathbf{C} \\ & \mathbf{f} \geq \mathbf{0} \end{aligned}$$

where $\underline{\mathbf{A}}$ is the incidence matrix for $\underline{D}$ has the same optimal solution that MAX-FLOW does on D. In fact, it should be clear that there is a $1-1$ correspondence between feasible solutions $\mathbf{f}$ for MAX-FLOW and feasible solutions $\underline{\mathbf{f}}$ for $\mathcal{P}$ where the two are the same in all edges except $\underline{e}$, and $\underline{\mathbf{f}}_{\underline{e}} = k$ if and only if the number of clams $\mathbf{f}$ can move from s to t is k. Since minimizing the cost equated to maximizing this value of k, the two will have the same optimal solutions.

Of course we now know two algorithms which solve the general network flow problem, but one is more "graph theory" oriented — the negative cost cycle algorithm (see Section 9.3) — so let's use that to find a "graph theory" oriented solution to the original MAX-FLOW problem. Recall that the whole point of the negative cost cycle algorithm is to find negative cost, unsaturated

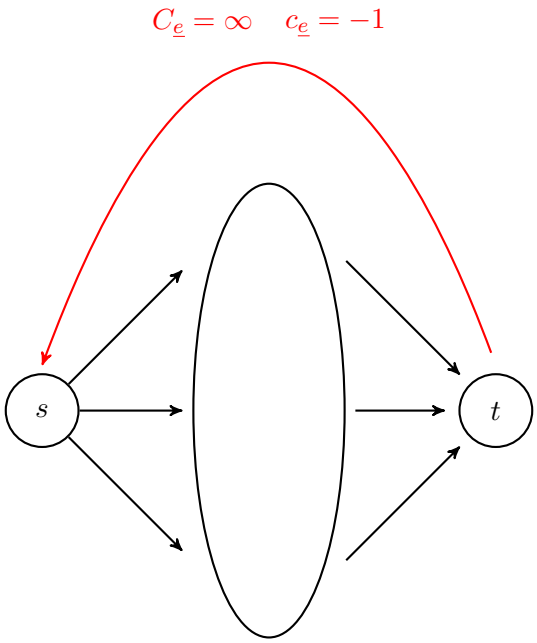

Figure 10: Reformulation of max flow as a network flow problem by adding edge $\underline{e}$ (in red).

cycles. Well, we know what *negative* cycles look like in $\underline{D}$ — they are the ones that contain $\underline{e}$. Furthermore, $\underline{e}$ can never be saturated (it has infinite capacity), so a given cycle that contains $\underline{e}$ will be unsaturated if and only if the corresponding $s - t$ path is unsaturated in D. And in order to hide the fact that we used the negative cost cycle algorithm, we will give these paths a new name.

An *augmenting path* in D is a set of edges in D that form a path from s to t in the underlying graph for which

- $f_e < C_e$ for all forward going edges e
- $f_e > 0$ for all backward going edges e

Lemma 39. *A collection of edges form an augmenting path if and only if it forms an unsaturated cycle when $\underline{e}$ is added.*

This leads to the following algorithm, known as the *Ford–Fulkerson* algorithm:

1. Assume you have a feasible flow $\mathbf{f}$ (you can start with $\mathbf{f} = \mathbf{0}$)
2. Find an augmenting path
   (a) If no augmenting path exists, terminate.
   (b) Otherwise, add the augmenting path to $\mathbf{f}$
3. Iterate.

And what is nice about this algorithm is that any graph theorist could understand it without having to know anything about linear programming[85]. The one remaining question is how one goes about finding augmenting paths, and the historical answer to this is via a *residual graph*.

[85] Of course the fact that it is equivalent to the negative cost cycle algorithm is useful for us because it means it will have exactly the same strengths and weaknesses.

Given a digraph $D = (V, E)$ with capacities $\mathbf{C}$ and a flow $\mathbf{f}$, the *residual graph* is the graph $D' = (V, E')$ — so same vertices — and where $(i, j) \in E'$ whenever (at least) one of the following conditions hold:

- The edge $(i, j) \in E$ and $f_{(i,j)} < C_{(i,j)}$
- The edge $(j, i) \in E$ and $f_{(j,i)} > 0$

In other words, $(i, j) \in E'$ if and only if there is an edge from i to j that can contribute to an alternating path. Note that every edge in E will give at least one edge in E' (possibly pointing in the opposite direction) and it may give two edges in E' (going both directions) if the flow value is not 0 but is also not at capacity. Furthermore, it should be clear that alternating paths in D turn into directed paths in D' (and vice versa), so we can find alternating paths by simply looking for directed paths in D' (which is much easier). We will say that a vertex $v \in V$ is *reachable from* s (or simply just "reachable" since s doesn't change) if there exists a directed path[86] in D' from s to v. This gives the following lemma:

Lemma 40. *There exists an augmenting path in D if and only if t is reachable in D'.*

Now assume that D does not have an augmented path, and let R denote the collection of vertices that are reachable. Clearly $t \notin R$, so R forms an $s - t$ cut, suggesting that we should go back and look at $s - t$ cuts.

10.2 `MIN-CUT`

For the moment, let us make a slight detour to the other problem we saw on the problem set. The `MIN-CUT` problem has the same set up as the `MAX-FLOW` problem, but now you are looking for the $s - t$ cut with the greatest capacity. The reason these two problems are linked is that cuts act as certificates for flows (and vice versa). To see why, let K be an $s - t$ cut defined by the set S ($s \in S$ and $t \notin S$). In order to get clams from s to t, you (in the process) need to get them from S to $\overline{S}$. And the maximum amount of clams you can send from S to $\overline{S}$ is the capacity of K. Hence for any flow $\mathbf{f}$ and any cut K, we have

$$\text{value}(\mathbf{f}) \leq \text{capacity}(K)$$

and so the two problems have a *weak duality* relationship. The obvious question to ask is whether it is a strong duality relationship, and the answer is (famously) yes.

Theorem 41 (`MAX-FLOW` / `MIN-CUT` Theorem)**.** *Let $D = (V, E)$ be a digraph with a single source s and single sink t. Given a vector of nonnegative capacities on E, the maximum possible $s - t$ flow allowed by these capacities is equal to the minimum capacity of an $s - t$ cut.*

To get a feeling for the proof, let me mention that the residual graph gives us a perfect candidate for a "'dual solution" (that is, a cut which has the same value that the flow $\mathbf{f}$ does). Let $\mathbf{f}$ be a flow in D which has no augmenting path and let

$$R = \left\{v \in V : v \text{ is reachable from } s \text{ in } D'\right\} \tag{28}$$

where D' is the residual graph. By definition, $s \in R$, and since $\mathbf{f}$ has no augmenting path, $t \notin R$, so R defines a unique $s - t$ cut. Because of the way R was built, it must be that

- all edges e going from R to $\overline{R}$ must have $f_e = C_e$, and

[86] Only forward edges can be used.

- all edges e going from $\overline{R}$ to R must have $f_e = 0$.

The goal will be to show that the value of the flow $\mathbf{f}$ is the capacity of R (so this cut does, in fact, provide a certificate). One can do this by analyzing the Ford–Fulkerson algorithm[87], but we will find it more instructive to use linear programming duality.

10.3 Proof of the MAX-FLOW / MIN-CUT theorem

We started by looking at the dual of the linear program defined in Section 10.1:

$$\begin{aligned} \mathcal{P} = \min \quad & -f_{\underline{e}} & & \\ \text{s.t.} \quad & \underline{\mathbf{A}}\mathbf{f} & = \mathbf{0} & \quad (1) \\ & \mathbf{f} & \leq \mathbf{C} & \quad (2) \\ & \mathbf{f} & \geq \mathbf{0} & \end{aligned}$$

where we associate dual vector $\boldsymbol{\lambda}$ to the equality constraints (1) and dual vector $\boldsymbol{\mu}$ to the capacity constraints (2)[88].

$$\begin{aligned} \mathcal{D} = \max \quad & \boldsymbol{\lambda} \cdot \mathbf{0} & + \quad & \boldsymbol{\mu} \cdot \mathbf{C} & \\ \text{s.t.} \quad & \boldsymbol{\lambda}^\intercal \underline{\mathbf{A}} & + \quad & \boldsymbol{\mu}^\intercal & \leq \mathbf{d}^\intercal \\ & & & \boldsymbol{\mu} & \leq \mathbf{0} \end{aligned}$$

where $\mathbf{d}$ is 0 except in the entry $d_{\underline{e}}$, where it is -1. Note that $\boldsymbol{\mu}$ has an entry for every edge (except $\underline{e}$ which is unconstrained) and $\boldsymbol{\lambda}$ will have an entry for each vertex v (similar to what we saw in Section 9.1.1). Hence the constraints become

$$\lambda_i - \lambda_j + \mu_{(i,j)} \leq 0 \qquad \text{for all edges } (i,j) \neq \underline{e}$$

along with the extra constraint $\lambda_t - \lambda_s \leq -1$ (coming from $\underline{e}$). In theory, the λ_i are free, but it should be clear that $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda} + \theta\mathbf{1}$ give the same solution (for any real value θ)[89]. So without loss of generality, we can normalize things so that $\lambda_t = 0$. Now if we assume that $\text{opt}(\mathcal{P}) \neq 0$, then it must be that $f_{\underline{e}} \neq 0$ and so (in particular) the constraint $f_{\underline{e}} = 0$ will not be an active constraint in $\mathcal{P}$. So by complementary slackness, the dual constraint

$$\lambda_t - \lambda_s \leq -1$$

must be active, which means we must have $\lambda_s = 1$.

Now when it comes to optimizing, we want to make $\boldsymbol{\mu} \cdot \mathbf{C}$ as big as possible, which means we want the $\mu_{(i,j)}$ to be as small (in absolute value) as possible. If $\lambda_i \leq \lambda_j$, then we can set $\mu_{(i,j)} = 0$ and still be feasible, so we can think of $\mu_{(i,j)} C_{(i,j)}$ as a penalty we must pay for having λ_i bigger than λ_j. Furthermore, we have the (incredibly nice) result that you proved on the problem set — since the cost function of $\mathcal{P}$ has only integer entries, there must exist an optimal solution to the dual that has *integer values*. From there, it is not hard to convince yourself that an optimal integer solution with $\lambda_s = 1$ and $\lambda_t = 0$ must have all $\lambda_v \in \{0, 1\}$ (since otherwise you are creating larger gaps (and therefore larger penalties) than you need.

[87] This is how it was done historically.

[88] To be clear, there is really just one dual vector, but we are going to call the first part of it $\boldsymbol{\lambda}$ and the second part $\boldsymbol{\mu}$ because that will make life easier.

[89] This again comes from the fact that $\underline{\mathbf{A}}$ is not full rank, but this time showing up in the dual. In fact, if we had done all of this with a truncated incidence matrix $\underline{\tilde{\mathbf{A}}}$ formed by removing vertex v, then the dual would be equivalent to $\mathcal{D}$ with the added constraint $\lambda_v = 0$.

As we saw previously, such vectors are in 1-1 correspondence with $s-t$ cuts, and it is easy to check that the resulting cost of such a vector[90] is the capacity of the cut that it corresponds to. In other words, $\mathcal{D}$ is a formulation of the `MIN-CUT` problem, and so strong duality tells us that the two have the same solution.

10.4 Wait... what just happened!?

The last thing I mentioned in class was that when you look at it the right way, the proof can sometimes look super easy (perhaps too easy) and this can make you think that this sort of "combinatorial duality" might happen a lot. In fact, that is NOT the case — it is not common for a combinatorial problem like `MIN-CUT` to have a strong dual. In particular, recall one of our favorite tricks that we like to do — turn a min problem into a max problem by multiplying the cost by -1. In theory, I should be able to solve the `MAX-CUT` problem this way — simply multiply the cost function in `MIN-CUT` by -1, take the dual (to get a `MIN-FLOW` problem) and argue that this has the same optimal value as the primal (which we can solve using Ford–Fulkerson).

The issue with this "logic" is that `MAX-CUT` is a well-known NP-complete problem, so something must be going horribly wrong in this process. For those paying attention, you can see (on the surface) what the error is — the thing we multiplied by -1 is the capacity vector, which (as we said in the original problem definition) needs to be a nonnegative vector. You would not be alone if you thought that the bit about $\mathbf{C}$ needing to be nonnegative was simply due to its interpretation as a graph theory problem. But it actually goes *much* deeper than that[91] — something we will see again in our next (and final) problem: `SHORTEST-PATH`.

10.5 References

1. Bertsimas and Tsitsiklis: Section 7.5.
2. The video example of Ford–Fulkerson (on Moodle)

[90] Really the cost comes from $\boldsymbol{\mu}$, but it is not hard to see that the optimal solution will require $\mu_{(i,j)} = -1$ if and only if $\lambda_i = 1$ and $\lambda_j = 0$ (and 0 otherwise) so everything becomes (essentially) a function of $\boldsymbol{\lambda}$.

[91] In particular, it means that our easy starting solution $\mathbf{f} = \mathbf{0}$ is no longer feasible!

11 Week 11: Intro to Shortest Path and Bellman–Ford

This week we will look at another problem that is related to network flows — the *shortest path problem*. This problem is quite easy to state and it should be clear why it would be important — you want to get from point A to point B and you want to do so with as little cost as possible. The fact that this problem is easy should not be a surprise, either, since this is something that we as humans tend to do naturally (when looking at a map, for example). However, it will be interesting to see how we can turn this natural sense of what we would do into an algorithm that (a) we can prove is optimal, and (b) is fast.

The inspiration for an algorithm will (as before) come from linear programming. To solve the actual "shortest path" problem, we would put costs of $+1$ on the edges. But more generally, we might want to solve the "minimum cost path" problem where we now allow arbitrary costs on the edges. When we start to allow arbitrary costs on edges, however, we need to be careful — just like before[92] we are dangerously close to NP-complete territory.

In particular, what would happen if we set all of the costs to be -1? Then the "minimum cost path" would be one that is as long as possible. In particular, it would be a Hamiltonian path[93] (if one were to exist). The issue is that the problem of determining whether a graph contains a Hamiltonian path is NP-complete, and so we should not expect to be able to solve it.

One obvious remedy is to restrict to having nonnegative edges, however we can be a bit more clever. If we think about how we might turn "minimum cost path" into an LP, we would quickly see that the "path" part is an issue — forcing every vertex to be used at most once is a a very "binary-programmish" kind of constraint. Hence one idea that we could try is to weaken the problem in a slightly different way — to look for minimum cost *walks* instead of *paths*.

Let's go back to our "bad example" and see what happens if we allow walks instead of paths. If we were to set all of the edges to -1 and look for a minimum cost walk, then we would quickly run into trouble if we found a cycle in the graph — going around that cycle over and over (and over) would just cause the cost to go down (forever). Furthermore, if all of the edges are positive, then it should be clear that the "minimum cost walk" (MCW) problem has an optimal solution which is a path (you can turn a walk into a path by removing some of the edges in the middle and — when those edges have nonnegative cost — your total cost can only go down). So the idea of looking for walks instead of paths seems to be a good compromise.

Lastly, it should be mentioned that there are lots of different flavors of MCW problems — the one you solve on a daily basis (where you need to get from A to B in as little time as possible) is known as the `ONE-TO-ONE` problem. To make things more interesting, we will consider two somewhat more complex problems: the `ALL-TO-ONE` problem (where we have a target vertex v and want to build a vector of telling us the values of a MCW from u to v for all vertices u) and the `ALL-PAIRS` problem where we want to build a matrix of the values of the MCWs between any two pairs of vertices. Obviously if we can solve the `ONE-TO-ONE` problem, we can solve these other two problems by just iterating the solution to the `ONE-TO-ONE` problem. The goal will be to solve both problems in faster (and more clever) ways.

Two more things: while we have already established the problem we are going to look at regards weighted walks and not paths, the historical way that people refer to this problem is as the "shortest path problem" so look out for that if you use any sort of resource other than these notes[94]. Secondly, unlike the previous sections, when we talk about paths/cycles/walks in this section, we will always be referring to the *directed* kind (where you can only use forward-going

[92] And this will be the case with almost any combinatorial optimization problem.

[93] A Hamiltonian path is a path of length $n-1$ on an n vertex graph.

[94] Including me — I have a habit of saying "shortest path" when I am really talking about MCWs.

edges)[95]. Just to make it clear (for when you are studying and can't remember which section has directed stuff and which one had undirected stuff), let me put it in a box.

> In Sections 11 and 12 (and these sections only), the terms path/walk/cycle will mean *directed* path/walk/cycle.

11.1 ALL-TO-ONE

We start with the ALL-TO-ONE problem. Formally, we are given a digraph $D = (V, E)$ and a cost vector $\mathbf{c}$ on the edges. We will assume that $|V| = n$ and that v_n is the "destination" vertex (the one we wish to find out how to get to). One obvious idea is to try to turn this into a network flow problem. In particular, let us give each of the vertices a unit of flow and force it to send that flow to v_n. That is $\mathbf{b}$ is defined as

$$b_i = \begin{cases} -(n-1) & \text{if } i = n \\ 1 & \text{otherwise} \end{cases}$$

Then each of the vertices should find its own MCW to v_n. Hence the linear program becomes one that we are familiar with:

$$\begin{aligned} \mathcal{P} = \min \quad & \mathbf{c} \cdot \mathbf{f} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{f} = \mathbf{b} \\ & \mathbf{f} \geq \mathbf{0} \end{aligned}$$

which has dual (see Section 9.1.1):

$$\begin{aligned} \mathcal{D} = \max \quad & -(n-1)\lambda_n + \sum_{i \neq n} \lambda_i \\ \text{s.t.} \quad & \lambda_i - \lambda_j \leq c_{(i,j)} \qquad \text{for all } (i,j) \in E \end{aligned}$$

One thing to notice (something we have seen before) is that for every feasible solution $\boldsymbol{\lambda}$ we have $\boldsymbol{\lambda} + \theta\mathbf{1}$ is also feasible (for all real numbers θ) and the cost is the same. Hence we can assume $\lambda_n = 0$, which means we can rewrite $\mathcal{D}$ as[96]

$$\begin{aligned} \tilde{\mathcal{D}} = \max \quad & \sum_{i \neq n} \lambda_i \\ \text{s.t.} \quad & \lambda_i - \lambda_j \leq c_{(i,j)} \qquad \text{for all } (i,j) \in E \\ & \lambda_n = 0 \end{aligned}$$

Now let's use our knowledge of the simplex method — we know that if there exists an optimal solution, then that optimal solution is a tree solution. Furthermore, this tree solution must have a special form (because of the linear program we are doing):

An *inrooted tree* is a spanning tree with a special vertex (called the root) such that all edges are pointed towards the root. Given what the initial problem is (everyone is trying to get to v_n), it should be clear that the optimal tree solution will be an inrooted tree. Furthermore, the edges of the inrooted tree are going to be the constraints that are active in the solution, which means λ_i^* is going to be the sum of the costs of the edges in the tree that are between v_i and v_n (in other words, the cost of the path!). This is summed up in the following lemma:

Lemma 42. *Let $\boldsymbol{\lambda}^*$ be an optimal solution to $\mathcal{D}$ with $\lambda_n^* = 0$. Then the MCW from vertex v_i to vertex v_n has total cost λ_i^*.*

[95] This only makes the algorithm more versatile — if you want to allow someone to go both ways on an edge, you can simply put directed edges there.

[96] Note that this is exactly what we would have gotten if we had formed $\tilde{\mathcal{P}}$ by removing row n from the incidence matrix and then taken the dual.

11.2 Bellman equations

Let's think about the process of constructing $\boldsymbol{\lambda}^*$ from a tree solution T. If v_i is a leaf, then the tree solution tells us that

$$\lambda_i^* = c_{(i,k)} + \lambda_k^*$$

where (i,k) is the single edge in T that is incident with vertex v_i. So if we were to take a different edge leaving v_i (say (v_i, v_t) where $t \neq k$) then the fact that this is an optimal tree solutions gives us an inequality. That is:

$$\lambda_k^* + c_{(i,k)} \leq \lambda_t^* + c_{(i,t)}$$

Why? Because if $\lambda_t^* + c_{(i,t)}$ had been smaller, then that would have given us a better solution! Hence every leaf vertex v_i must satisfy

$$\lambda_i^* = \min_{k:(i,k)\in E} \{c_{(i,k)} + \lambda_k^*\}. \tag{29}$$

By pulling off the leaves inductively, we can show that this must be the case for all vertices in the graph. That is, if $\boldsymbol{\lambda}^*$ is an optimal solution to $\mathcal{D}$, then (29) must hold for all vertices v_i. These are called *Bellman's equations* because Bellman came up with the idea of using this characterization as a way to solve the problem (with Ford from Ford–Fulkerson).

Lemma 43 (Bellman)**.** *Let $\boldsymbol{\lambda}^*$ be a solution to Bellman's equations for which λ_i^* is the cost of (any) walk from v_i to v_n for all i. Then $\boldsymbol{\lambda}^*$ is a vector of MCWs.*

Proof. It should be clear that any finite solution to Bellman's equations is dual feasible, however such a solution may not correspond to actual walks in the graph (see Figure 11). If the solution DOES match the costs of actual walks in the graph, then those walks can be used to construct a feasible solution to the primal with the same total cost (this will be on the Problem Set). Hence we have a primal feasible solution and a dual feasible solution each with the same cost, which (by weak duality) implies both solutions are optimal for their respective problems. □

So the goal would be to (rather than mess around with simplex), simply try to find good solutions to these equations. But in order to find good solutions, one must find *any* solutions. Bellman's idea for doing this comes from the theory of dynamical systems, where one studies (among other things) what happens when you have a function $f : \mathbb{R}^n \to \mathbb{R}^n$ and a starting point $\mathbf{x}$ and you just keep doing f over and over again. That is, how does the sequence

$$\mathbf{x}, f(\mathbf{x}), f(f(\mathbf{x})), f(f(f(\mathbf{x}))), \ldots$$

behave? There are three possibilities:

1. The sequence converges to a point $\mathbf{x}^*$ after a finite number of iterations.
2. The sequence converges to a point $\mathbf{x}^*$, but requires an infinite number of iterations.
3. The sequence doesn't converge.

The interesting possibility in our case is the first one — if we assume that the sequence converges to $\mathbf{x}^*$ after a finite number of iterations, then (by definition) it must be that $\mathbf{x}^*$ is a solution to the equation

$$\mathbf{x} = f(\mathbf{x}).$$

So Bellman suggested considering the following function: $F(\mathbf{x}) = (F_1(\mathbf{x}), \ldots, F_n(\mathbf{x}))$ where

$$F_i(x_1, \ldots, x_n) = \begin{cases} 0 & \text{if } i = n \\ \min\limits_{k:(i,k)\in E} \{c_{(i,k)} + x_k\} & \text{otherwise} \end{cases}$$

It is not hard to compute $F(\mathbf{x})$ from $\mathbf{x}$, and if the sequence converges after a finite number of steps, then we would left with a solution to Bellman's equations (exactly what we were trying to find). As noted above, we need to be careful because there could be solutions which don't correspond to actual walks in the graph (see Figure 11). However, if we are able to find a solution that corresponds to actual walk cost values, then Lemma 43 guarantees that this will be an optimal solution. This leads to what is known as the *Bellman–Ford algorithm*.

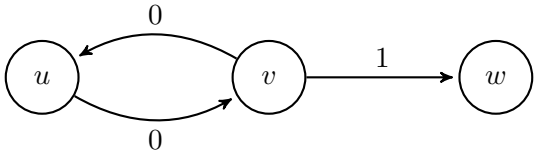

Figure 11: A graph for which $\mathbf{x} = F(\mathbf{x})$ has solutions $(0, 0, 0)$ and $(1, 1, 0)$. Only the second one corresponds to a valid list of MCWs.

11.3 Bellman–Ford Algorithm

The Bellman–Ford algorithm is an iterative algorithm that one can view as a series of improving estimates on the values of the MCWs. The values at every round will be constructed in a way that corresponds to actual walks in the graph. That way, if we do find a solution to Bellman's equations, we will know it is the optimal one. We define $B_i(t)$ to be

$$B_i(t) = \{\text{the smallest total cost of any walk from } v_i \text{ to } v_n \text{ using at most } t \text{ edges}\}.$$

where we set $B_i(t) = \infty$ if getting from v_i to v_n using at most t edges is impossible. In particular, $B_n(t) = 0$ for all t and $B_i(t) \leq B_i(t-1)$ for all i and t.

Lemma 44. *For all $i \neq n$,*

$$B_i(t+1) = \min_{k:(i,k)\in E} \{c_{(i,k)} + B_k(t)\}.$$

Proof. Let w be the smallest cost walk from v_i to v_n using at most $t+1$ edges. Since $i \neq n$, w must take at least one step, and so and that first step must take us to a vertex v_j for which $(i, j) \in E$ (since that is how we make steps). The cost of taking this first step is then $c_{(i,j)}$.

What does the rest of w look like? Well, we have to (somehow) get from v_j to v_n and we can only use t edges (since we already used one). Furthermore, the fact that w has minimum possible cost means that this subwalk needs to have minimum possible cost. However, we have a name for the value of "the smallest cost walk from v_j to v_n using at most t edges": $B_j(t)$. Hence the cost of w must be

$$B_i(t+1) = c_{(i,j)} + B_j(t).$$

But now suppose $(i, k) \in E$ for some $k \neq j$. That means our first step could have been to v_k, so why didn't w take that one? If w had taken it, it would have led to a walk with total cost

$$c_{(i,k)} + B_k(t)$$

so the fact that w (an optimal walk) didn't take that path means that it must not have been better than the one through v_j. In other words, the fact that w is optimal and w's first step is to v_j guarantees that

$$c_{(i,j)} + B_j(t) \leq c_{(i,k)} + B_k(t)$$

for any other edges $(i, k) \in E$. Hence

$$B_i(t+1) = \min_{k:(i,k)\in E} \{c_{(i,k)} + B_k(t)\}.$$

as required. □

On the Problem Set, you will prove the following:

Lemma 45. *Assume $\mathcal{P}$ has a feasible solution*[97] *and let $\mathbf{B}(t) = (B_1(t), \ldots, B_n(t))$. Then there exists a negative cost cycle in $\mathcal{P}$ if and only if $\mathbf{B}(n) \neq \mathbf{B}(n-1)$.*

This tells us two things:

- If $\mathcal{P}$ is feasible and there is a negative cost cycle, we will know about its existence in $n = |V|$ rounds.
- If $\mathcal{P}$ is feasible and there is no negative cost cycle, then $\mathbf{B}(n-1)$ satisfies $\mathbf{x} = F(\mathbf{x})$. Furthermore, since these values correspond to actual walks (this is how we constructed them) $\mathbf{B}(n-1)$ will be the vector of MCW values.

The one remaining case — when $\mathcal{P}$ is not feasible — will also be revealed in n rounds as this will result in $B_i(n) = \infty$ for some i. Hence in every situation we will need only n rounds to get a solution. Now if you look at each round, the number of comparisons you need to do in order to get from $\mathbf{B}(t)$ to $\mathbf{B}(t+1)$ is at most the number of edges in the graph (each edge only has one head, and that edge will only need to be considered for calculating $B_{i+1}(t)$ if its tail is i). Hence this algorithm runs in $O(|V||E|)$ time.

11.4 References

1. Bertsimas and Tsitsiklis: Section 7.9
2. Video example of Bellman–Ford (on Moodle)

[97] We need this assumption because (technically) one could have $\mathbf{B}(n) = \mathbf{B}(n-1)$ and still have a negative cost cycle if there was no way to get from the negative cost cycle to v_n (as then distances would all be ∞ and never change).

12 Week 12: Dijkstra and All-Pairs Shortest Path

The following is a continuation from the previous section. If it doesn't make sense, read the paragraph directly before Section 11.1.

> In Sections 11 and 12 (and these sections only), the terms path/walk/cycle will mean *directed* path/walk/cycle.

Last time we saw the Bellman–Ford algorithm for solving the `ALL-TO-ONE` problem. It was robust in that

- if there is a negative cost cycle, it would detect this
- if there is no negative cost cycle, it would find all of the minimum cost walks (which turn out to be paths)

Furthermore, it ran in $O(|E||V|)$ time. Here we present another algorithm for solving the `ALL-TO-ONE` problem known as *Dijkstra's algorithm*. It is significantly faster than Bellman–Ford, but it is less robust — in fact, it will only work on a network where all edge costs are nonnegative.

The reason negative edge costs matter so much is that they can turn what looks like a good decision into a "not so good" one. For example, imagine you had a 3-vertex graph and you were doing Bellman–Ford. In round 1 you would looking at all paths containing one edge, and you saw Figure 12.

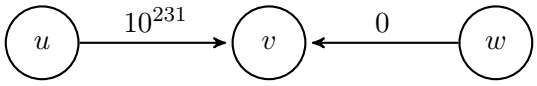

Figure 12: Round one of a Bellman–Ford.

It seems like going from $w \to v$ along the edge (w, v) is a good move. So good, you might think there was no way that knowing the third edge could change anything. But then you get to round 2 and you see Figure 13 and you realize that (w, v) edge isn't so great after all.

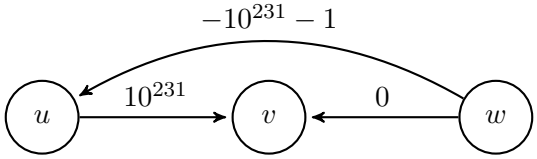

Figure 13: Round two of a Bellman–Ford.

However, if we did not allow negative edges, then your initial feeling that (w, v) was going to be the best would have been right. So, for example, take a digraph like the one in Figure 14 and look at all of the in-edges for v_n. v_3 has the smallest cost so we *know* that the path (v_3, v_n) is going to be the cheapest option.

In fact, we know it *so well* that there really isn't any point in keeping v_3 around any more. Certainly all of the out-edges at v_3 (to other vertices) are useless because why would we ever use them? So we can throw those away. BUT you might say, we still need to keep the in-edges at v_3 in case a future minimum cost walk needed to go through v_3. And you would be right, but any minimum cost walk that goes through v_3 will have to take the edge (v_3, v_n) in the next move, so we can just update those edges (adding the cost of (v_3, v_n) and having them go straight

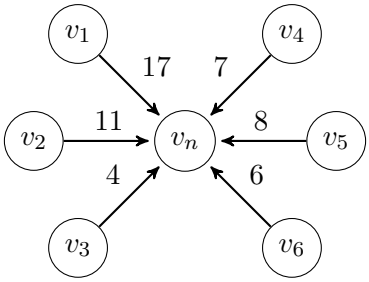

Figure 14: Round one of a Bellman–Ford with no negative edges.

to v_n) as in Figure 15. This will create multiple edges, but that is OK — we really only really care about the edge with the smallest cost, so we can throw the others out.

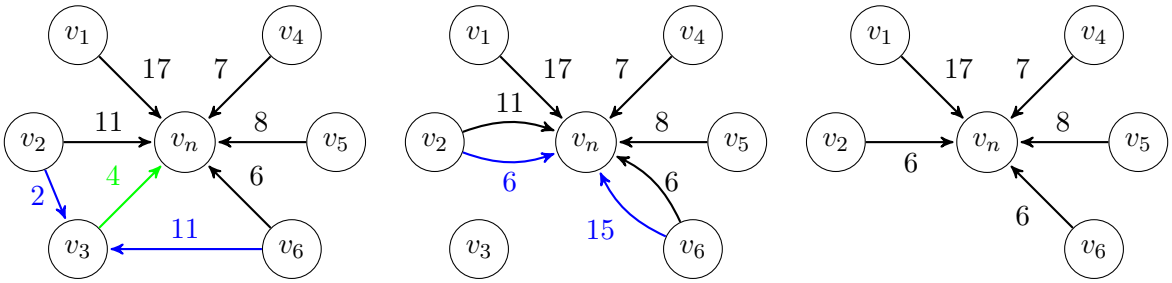

Figure 15: Eliminating the smallest cost path of length 1 (one round of Dijsktra).

Once we have recorded the value for v_3 and rerouted any edges that go through v_3 (adjusting their costs accordingly), v_3 is no longer relevant, so we can simply remove it from the graph. This leaves us with a smaller graph and then we can iterate. *BUT* you say, even though we are getting the combined path lengths, we are losing the knowledge of what the actual paths are. And (again) you would be right, but fortunately we don't need to know the actual paths because you showed in the Problem Set how one can reconstruct the paths from the vector of minimum cost values.

Now, let's look at how efficient Dijkstra's algorithm is. We had to

- Find the smallest in-edge to v_n (which could take up to $|V|$ comparisons)
- Reroute all of the in-edges that were going to the closest vertex (which could take up to $|V|$ comparisons)
- Reduce all of the double edges by picking the smallest one (which could take up to $|V|$ comparisons)

So all in all, each round takes $O(|V|)$ steps and there were $O(|V|)$ rounds for a total of $O(|V|^2)$.

So for graphs with a lot of edges, this will be much faster than Bellman–Ford but at the cost of not being able to have negative cost edges. In particular, if the graph is dense[98], then Bellman–Ford would take $O(n^3)$ times and Dijkstra $O(n^2)$.

One (obvious) way to solve `ALL-PAIRS` is by simply solving `ALL-TO-ONE` once for each vertex.

Doing Bellman–Ford each time would take $O(|V|^2|E|)$ time whereas Dijkstra would take $O(|V|^3)$. This is more or less the same when $|E|$ is small, but for graphs with many edges (cn^2 for some c), this is a big difference. Since the complexity of Dijkstra is independent of the

[98] A *dense* graph has a positive fraction of all possible edges (n vertices and more than cn^2 edges for some c).

number of edges, it will stay at $O(n^3)$, but Bellman–Ford (which does depend on the number of edges) will balloon to $O(n^4)$. In particular, doing Dijkstra on all n vertices is pretty much the same as doing Bellman–Ford for 1 vertex when you have a dense graph.

This might seem like a coincidence (and in most cases it would be, since there are TONS of $O(n^3)$ algorithms around and most of them have nothing to do with each other. But in *this* case, we can use this fact to get a more robust `ALL-PAIRS` algorithm. The idea is going to be straightforward — take a problem with negative costs on the edges and try to turn it into one that has nonnegative costs on the edges. Of course, this is not a trivial thing, since we want to change the costs in a way that doesn't change the solution! In particular, the obvious first attempt that many people make of "adding x to every edge" can change the solution, as can be seen in Figure 16. In fact, the easiest way to find out whether this is possible (and how) is to go back to linear programming.

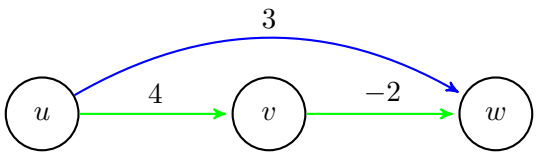

Figure 16: The minimum cost walk from v to w (in green) changes (to the one in blue) if we were to add 2 to all edges.

12.1 An interlude: linear programs with the same solutions

Let's pretend that you are interning for company X and for (whatever reason) they have given you a linear program in inequality standard form to solve:

$$\begin{aligned} \mathcal{P} = \max \quad & \mathbf{c} \cdot \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \quad \geq \mathbf{b} \end{aligned}$$

and so you solve it and get an optimal solution $\mathbf{x}^*$ and an optimal dual solution $\boldsymbol{\lambda}^*$ that you can use to certify the optimality. And then the company realizes they made a mistake — they were supposed to give you

$$\begin{aligned} \mathcal{P}' = \max \quad & \mathbf{c} \cdot \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \quad \geq \mathbf{b}' \end{aligned}$$

instead, so now you have to start all over again. Or do you? Are there situations where the solution to $\mathcal{P}$ can be used to get a solution to $\mathcal{P}'$ without having to redo the entire solution process?

The answer is "yes" but only when $\mathbf{b}'$ and $\mathbf{b}$ are related in a very particular way:

$$\mathbf{b}' = \mathbf{b} + \mathbf{A}\mathbf{y}$$

where $\mathbf{A}$ is the same matrix as in your linear programs and $\mathbf{y}$ is some fixed vector. To see why, try plugging this identity into $\mathcal{P}'$:

$$\begin{aligned} \mathcal{P}' = \max \quad & \mathbf{c} \cdot \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \quad \geq \mathbf{b} + \mathbf{A}\mathbf{y} \end{aligned}$$

and then substitute $\mathbf{z} + \mathbf{y} = \mathbf{x}$. Now the only difference between $\mathcal{P}$ and $\mathcal{P}'$ is the extra $\mathbf{c} \cdot \mathbf{y}$ in the cost function, but this is *a fixed quantity*, so we can simply calculate it:

$$\begin{aligned} \mathcal{P}' = \mathbf{c} \cdot \mathbf{y} + \max \quad & \mathbf{c} \cdot \mathbf{z} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{z} \quad \geq \mathbf{b} \end{aligned}$$

which (is except for a shift by fixed constant) the problem you already solved. And any vector that maximizes $\mathbf{c} \cdot \mathbf{x}$ will also minimize $17 + \mathbf{c} \cdot \mathbf{x}$ (or whatever $\mathbf{c} \cdot \mathbf{y} + \mathbf{c} \cdot \mathbf{x}$ happens to be). So the point is, we were able to change $\mathbf{b}$ without *really* changing the problem.

And just like everything else in linear programming, this has a dual form. The dual of inequality standard form is equality standard form, and the role of $\mathbf{b}$ is played by $\mathbf{c}$ in the dual. So if I was given the linear program

$$\begin{aligned} \mathcal{D} = \min \quad & \mathbf{c} \cdot \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

then I might try to find a $\mathbf{c}'$ so that the linear program is (essentially) the same. In particular, if

$$\mathbf{c}'^{\mathsf{T}} = \mathbf{c}^{\mathsf{T}} + \mathbf{y}^{\mathsf{T}}\mathbf{A}$$

for some fixed vector $\mathbf{y}$, then

$$\begin{aligned} \mathcal{D}' = \min \quad & \mathbf{c}' \cdot \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

is the same as

$$\begin{aligned} \mathcal{D}' = \min \quad & \mathbf{c} \cdot \mathbf{x} + \mathbf{y}^{\mathsf{T}}\mathbf{A}\mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

which is the same as (since $\mathbf{A}\mathbf{x} = \mathbf{b}$ in any feasible solution)

$$\begin{aligned} \mathcal{D}' = \mathbf{y} \cdot \mathbf{b} + \min \quad & \mathbf{c} \cdot \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

which is the same problem (just shifting the cost function by a fixed amount).

Why do we care? Well, remember that the problem we are trying to solve here (minimum cost path) was something we could phrase as a network flow problem (see Section 11.1). And furthermore, the dual of this linear program had a term of the form $\boldsymbol{\lambda}^{\mathsf{T}}\mathbf{A}$ in it (and is something we called an *induced flow* — see Section 9.1.1). So this says if we have a minimum cost walk problem with a cost vector $\mathbf{c}$, we can add an induced flow to $\mathbf{c}'$ and the resulting linear program will have the same optimal solution (that's crazy!). *In particular,* if we had a graph with costs on the edges $\mathbf{c}$ (where some of the c_i are negative) and if we could find an induced flow which, when added to $\mathbf{c}$, makes the (new) costs nonnegative, then we could use Dijkstra instead of Bellman–Ford and really speed things up when there are a lot of edges. So this is going to be our approach to the `ALL-PAIRS` problem.

12.2 `ALL-PAIRS`

So, by what we discussed in the previous section, our goal for solving `ALL-PAIRS` will be to find an induced flow that we can add to $\mathbf{c}$ to make it nonnegative. Assuming we can do this, we can then run Dijkstra's algorithm n times (one for each vertex as the destination) and solve the `ALL-PAIRS` problem. The second part will take $O(n^3)$ time, and since Bellman–Ford has the potential to take $O(n^3)$ time, this would be quite a success.

This leaves the question: can we find such an induced flow (and how long will it take)[99]? Let's start by writing down what it is we are trying to find: we want to find a vector $\mathbf{w}$ for which

$$\mathbf{c}^\intercal + \mathbf{w}^\intercal \mathbf{A} \geq \mathbf{0}.$$

The first good news is that this is something we can actually do because it is just *another linear program*. So if a solution does exist, we know we will be able to find it (although possibly using simplex which is quite slow). However this equation resembles something we have seen before: if we set $\boldsymbol{\lambda} = -\mathbf{w}$ then this becomes

$$\boldsymbol{\lambda}^T \mathbf{A} \leq \mathbf{c}^\intercal.$$

which might look more familiar because it is precisely the set of constraints from the dual network flow problem in Section 9.1.1. So it turns out any feasible solution to the dual of the `ALL-TO-ONE` program can be turned into the kind of induced flow we are looking for.

And it just so happens that we know an algorithm that

- can find a feasible solution to the dual of the `ALL-TO-ONE` program
- can also tell us if none exists (by finding a negative cost cycle)
- runs in $O(n^3)$ time[100]

Drum roll, please.... Bellman–Ford! Putting all of this together gives us our final `ALL-PAIRS` algorithm.

12.2.1 Johnson's algorithm

Assume we are given a graph $D = (V, E)$ and a cost vector $\mathbf{c}$ on the edges which may or may not have negative values.

1. Pick a vertex v_n as a destination and run Bellman–Ford.
   (a) If Bellman–Ford doesn't converge after n rounds, there is a negative length cycle (so stop).
   (b) If Bellman–Ford does converge after n rounds to a vector $\boldsymbol{\lambda}^*$, then $\boldsymbol{\lambda}$ is the vector of MCW values from each vertex to v_n.
2. Set
   $$\mathbf{c}'^\intercal = \mathbf{c}^\intercal - \boldsymbol{\lambda}^{*\intercal} \mathbf{A}.$$
   By Bellman's equations, $\mathbf{c}'$ is nonnegative.
3. Solve the `ALL-TO-ONE` problem for all other vertices using Dijkstra and the cost vector $\mathbf{c}'$.
4. Use your method from the Problem Set to turn each of the `ALL-TO-ONE` solutions into actual MCWs (with respect to $\mathbf{c}'$).
5. Those will be the same MCWs when you replace $\mathbf{c}'$ with $\mathbf{c}$, so calculate their values (under $\mathbf{c}$) and those are your answers.

[99] In particular, if we can do it in time faster than $O(|V||E|)$, then we will have made Bellman–Ford obsolete!

[100] Which is still not *fast*, but on the other hand, we know that we will already be spending $O(n^3)$ time running Dijkstra over and over, so a faster algorithm wouldn't help all that much.

Since every step takes $O(n^3)$ time, the entire algorithm has complexity $O(n^3)$.

The most striking thing about this algorithm is that there are parts that we likely never would have been able to come across without the linear programming formulation (and simplex). And yet we still managed to find an algorithm which doesn't need simplex. Somehow, the simple fact that simplex *could have solved the problem* gave us the information we needed to find a better way.

12.2.2 Floyd–Warshall

The last thing we saw was a way to solve the `ALL-PAIRS` problem using a "Bellman–Ford style" algorithm[101] which also takes $O(n^3)$ time, called the *Floyd–Warshall algorithm*. Instead of using the Bellman function

$$B_i(t) = \{\text{the smallest total cost of any walk from } v_i \text{ to } v_n \text{ using at most } t \text{ edges }\}.$$

it uses the function(s)

$$p^k_{i,j} = \{\text{the smallest total cost of any walk from } v_i \text{ to } v_j \text{ using only the vertices } v_1, \ldots, v_k \text{ as intermediate vertices }\}.$$

Essentially, there are two options at stage k:

1. The minimum cost walk at this stage uses vertex v_k — the resulting walk then has cost
$$p^{k-1}_{i,k} + p_{k,j}(k-1)$$
2. The minimum cost walk at this stage does not use vertex v_k — the resulting walk then has cost
$$p^{k-1}_{i,j}.$$

This leads to the recursion

$$p^k_{i,j} = \min\left\{ p^{k-1}_{i,j} \ , \ p^{k-1}_{i,k} + p^{k-1}_{k,j} \right\}.$$

This convinced us that, despite being somewhat difficult to construct, "Bellman–Ford style" (more commonly known as *Dynamic programming*) algorithms are quite useful, and so this will be the topic of the final class.

12.3 References

1. Bertsimas and Tsitsiklis: Section 7.9

[101] Also known as *Dynamic Programming*.