Linear Programming: A half-baked Guide

Olivia Kirk
Data Science and Knowledge Engineering
Maastricht University

June 15, 2020

Missing Topics

- Integer Linear Programming topics
- Graph theory topics

How to Use

This is was intended to be a refresher guide, but I just couldn't help myself and rewrote the entire damn course. ~~However because I am literally writing this days before the exam, and have another exam to study for on the same day~~ Despite my revisions, I am depressingly aware of how lacking these notes really are. ~~I'm sure this is a goldmine for typos...~~

A slight inconvenience to note is that I used the slides from the 2019 version of this course, so there may be some referencing issues.
There are no fully worked examples for examinable material, so I will have to request you look solutions Steven releases for the bonus assignments.
However, due to my obsessing, this should still prove to be a deeply valuable resource on a theoretical level as it is made from the perspective of someone who needs things spelt out for them! I also believe I tried to make things more explicit by reiterating key points when relevant as well as attempting to link a more graphical framework to the abstract content ~~for those of you who, like me, can only understand things through pretty pictures~~.

Linear programming is a beautiful course; and I'm not just saying that because I like licking boots, that's just a happy coincidence. Things will fit together very logically and elegantly *once* you understand them, so don't let any confusion or doubt get the better of you.

Contents

Why do we care about Linear Programming?

- They can solve a problem to optimality given the problem can be expressed linearly.
- They scaled-up well in terms of speed through mathematical tricks and shortcuts.
- Flexible application that spans many real-world problems which can be formulated as a linear program. If there's an optimisation problem, linear programming will likely be there (these places can be unexpected).
- Can generate additional information about the system, such as the system's sensitivity to changing the original input and the presence of multiple optima. This is useful for making decisions about how to deal with the real-life equivalent of the system.
- From a computer science perspective, they clearly separates *what* is wanted and *how* we achieve it.
- They often are generated as a sub-routine of other algorithms (a classic and cheat-y example of this is Integer Linear Programming (ILP).
- The theory behind linear programs is crucial for understanding certain theoretical mathematical fields, where linear programs may never written or solve but the theory is ever-present.
- This is a very simple form of mathematical programming, that lays the ground work for grasping more complex approaches e.g. non-linear programming, ILP)

1 Modelling Linear Programs

I'll assume you know this foundational knowledge and be brief here, especially because it's a bit too intuitive for me to explain through text alone. The following is the general form of a (maximising) Linear program where we try to maximise the *objective function* Z:

$$\text{Maximise:} \quad Z = f(x) = \mathbf{c}x$$

$$\text{subject to:} \quad \mathbf{A}x \leq \mathbf{b}$$

$$\text{where } x_i, \geq 0, \quad i \, \epsilon \, \mathbb{Z}, \quad i > 0, \quad i \leq n.$$

$$\mathbf{A} = \begin{matrix} \\ \text{m} \end{matrix} \left\{ \overbrace{\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & 1/2 & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{bmatrix}}^{\text{n}} \right., \quad x = \begin{matrix} \\ \text{n} \end{matrix} \left\{ \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \right., \quad \mathbf{b} = \begin{matrix} \\ \text{m} \end{matrix} \left\{ \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \right. \text{ and } \mathbf{c} = \overbrace{\begin{bmatrix} c_1 & c_2 & \cdots & c_n \end{bmatrix}}^{\text{n}}$$

where A is a matrix of width n for each decision variable and height m for each functional constraint. Moreover x is actually a length n column vector containing each x_i value.

Note that the m constraints of a system are known as **functional constraints**, while the constraint boundaries are both the functional constraints and the non-negativity constraints.

Without sacrificing generality, we can assume a linear program:

1. is a maximisation program,
2. has only non-negative decision variables
3. and all constraints are $\leq$

2 The Simplex Method

Recall from linear algebra, for the system $\mathbf{C}x = \mathbf{d}$, where $\mathbf{C}$ is an $m \times m$ matrix and $\mathbf{d}$ is a column vector of length m, the column rank is equal to the row rank which is then also equal to m. For those of you who forgot what rank means, it means this is the number of rows/columns that are linearly independent. Moreover this is equivalent to C being invertible, and C^{-1} existing.
If these conditions do not hold in a linear program, either no solutions exist or infinite may solutions exist. The Simplex method is only interested in unique solutions, also referred to as "corner-point feasible" solutions, which are used to "climb" up to the optimal solution.

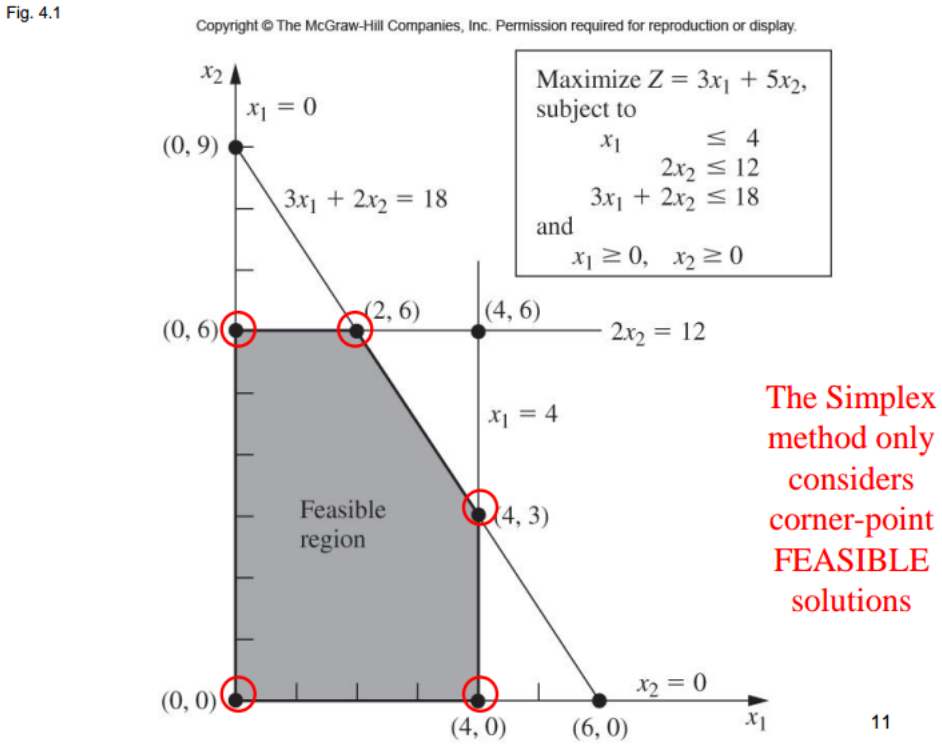

Figure 1: Visual on corner-point feasible solutions. Stolen very illegally from Steven's slide (Lecture 3, slide 14, 2019).

The Simplex method works by projecting solutions for systems of inequalities onto solutions for *higher dimensional* systems of *equalities*. What does means this mean? Well, as we can see from the picture above, there are only two dimensions which matches the number of decision variables (x_1 and x_2). We project by adding more variables to the system. In this case are called slack variables where the slack turns an inequality into an equality.
An example from Figure 1 is the constraint:

$$x_1 \leq 4 \quad \text{where} \quad x_1 \geq 0$$

we can convert this to an equivalent but higher dimensional form of

$$x_1 + x_3 = 4 \quad \text{where} \quad x_1, x_3 \geq 0$$

Through adding these slack variable, we create an *augmented form* of the original system. This augmented form will have 1 new slack variable for each functional constraint of the system. In terms of dimensions, the original system has n dimensions, which matches the number of decision variables, where the Simplex projects onto $m + n$ dimensions.

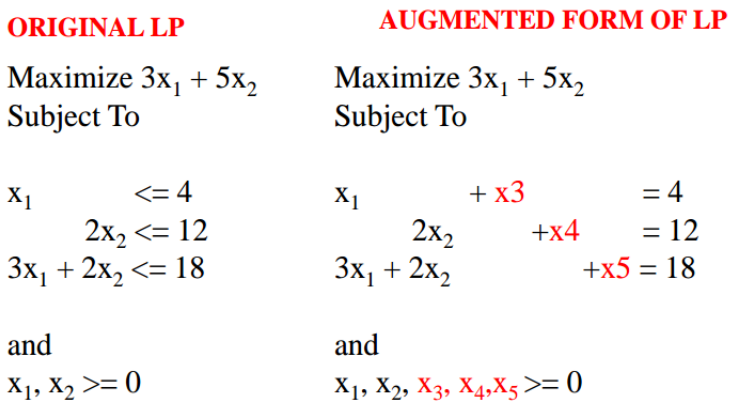

Figure 2: An original and augmented system side by side. Stolen from Steven's slide (Lecture 3, slide 24, 2019).

The augmented system can be expressed in **tableau form** as:

$$\left[\begin{array}{c|ccccc|c} & x_1 & x_2 & x_3 & x_4 & x_5 & RHS \\ \hline | & -3 & -5 & 0 & 0 & 0 & 0 \\ \hline x_3 & 1 & 0 & 1 & 0 & 0 & 4 \\ x_4 & 0 & 2 & 0 & 1 & 0 & 12 \\ z_5 & 3 & 2 & 0 & 0 & 1 & 18 \end{array}\right] \tag{1}$$

Note the very left column contains the *current basis* of the tableau. The Simplex method tends to start with the slack variables as the basis, where basic variables have 0 in the top row (which I will refer to as the Z row or as *the current objective function*). Also note that the objection function's values are now negative. This is because we rewrote $Z = 3x_1 + 5x_2$

into $Z - 3x_1 - 5x_2 = 0$.

An important concept we should talk about now is the tightness of a constraint. When $x_3 = 0$, that means the original inequality is satisfied by equality. Graphically, it means that x_1 is hugging the boundary of the inequality such as at $(4, 0)$ and $(4, 3)$ as well as the infeasible corner-point $(4, 6)$ in Figure 1. This concept is important because it tell us about the basis of the current Simplex tableau.

Back to linear algebra, because the augmented matrix has a rank of 3 due to the 0-1 pattern, there exists a subset of columns that are linearly independent. Note that each subset generates a single unique corner-point solution, but a corner-point solution may be generated by many subsets. To obtain a unique solution, we force 2, or more generally n variables to 0 to create our 3 x3 matrix, or more generally m xm . These n variables forced to 0 (and thus have a value of 0) are called **non-basic** variables. This is in contrast to the remaining m **basic variables**.
It is critical to note that basic variable CAN have a value of 0 as they only need to be non-negative. This is called *degeneracy* and we look at it in Section 2.2.

Speaking of generating corner-point feasible solutions, lets look at understanding them graphically.
Initially, we only had 2 dimensions so we used (x_1, x_2) as a coordinate system on Figure 1. However, we now have 5 dimensions so the coordinate system changes to $(x_1, x_2, x_3, x_4, x_5)$, where you can visualise the slack variables x_3, x_4 and x_5 on the same n-dimensional graph as *the distance from the function constraints to the current decision variable coordinates* (x_1, x_2).
As an example, the location $(0, 0)$ is equivalent to $(0, 0, 4, 12, 18)$, as the slack variables are carrying all the value. The basis is $\{x_3, x_4, x_5\}$, we have the n decision variables as non-basic and forced to 0, where they are tight at their constraint boundaries $x_1 \geq 0$ and $x_2 \geq 0$.

Some 'basic' terminology ~~ahaha~~ before we go any further
Basis, the basic variables of an augmented system e.g. $\{x_3, x_4, x_5\}$.
Basic solution, the values of the basis e.g. $(0, 0, 4, 12, 18)$.
Basic feasible solution (BFS), a basic solution that doesn't violate the constraint boundaries, a synonym of corner-point solution.

One way to view the Simplex method is that it iteratively pivots between adjacent corner-points until it reaches optimality. We can see this when the top row is non-negative for a maximisation program. But what is an adjacent corner-point? Typically, excluding degenerate cases, two adjecent corner-points share $n - 1$ non-basic variables/tight constraint boundaries.
This means when we pivot, one non-basic variable enters the basis and equally a basic variable leaves. These are known as the entering and leaving variables.

2.1 Applying the Simplex Method

1. Convert the original system into the augmented system in its tableau form, as in Equation 1.
2. If the objective function is purely non-negative in the tableau, you're already at optimality and can go home ~~exam over!~~.
3. Choose the entering basic variable x_e by selecting the column with the most negative value in the objective function (because we want to maximise it).
4. Choose the leaving basic variable e_l through the minimum ratio test.
5. Apply Gaussian elimination to make the x_e column follow the 0-1 pattern and thus removing it from the objective function.
6. If the current objective function contains any negative values (for a maximising program), return to *Step 3*. Otherwise you've reached optimality!

What if there is no *leaving basic variable*?
When there are no minimum ratio values strictly greater than 0, then either there is an unbounded optimum or (more likely) there is a mistake in the model.

What if there is a tie for either the entering or the leaving basic variable?
Pragmatically, you can just choose either and then look at some cute sloth pictures. Ideally, we address this by applying Bland's rule which we learn about when looking at the complex phenomenon known as *degeneracy* discussed in Section 2.2.

Figure 3: Lalala! Lala lala! (Lecture 4, slide 33, 2019).

2.1.1 The Minimum Ratio test

Given the entering basic variable x_e, finding the leaving basic variable x_l can be done by reforming each functional constraint to

$$x_i \leq RHS_i - \alpha_i x_e$$

where RHS_i is the right hand side value of constraint i, α_i is the scalar for that constraint's x_e value and yes, we just drop the other stuff for the test. It is important to note that we

only do this for the constraints that have x_e ***coefficients*** α ***strictly greater than 0***. We can ignore the constraint otherwise. After this, we reform once more into:

$$M_i = \frac{RHS_i}{\alpha_i}$$

where we want to find the smallest, **non-negative** M_i. This is because we want the value we can *force to 0* the quickest, hence eliminating all values below it.
In the earlier example tableau from Equation 1, we can see x_e is x_2. Next we rewrite the functional constraints into:

$$x_3 \leq 4 - (0)x_2$$
$$x_4 \leq 12 - (2)x_2$$
$$x_5 \leq 18 - (2)x_2$$

where we can drop the top constraint (because it has an α that is non-positive) and then export the remaining two into the forms of

$$M_4 = \frac{12}{2} = 6, \quad M_5 = \frac{18}{2} = 9.$$

from which we can see that $M_4 < M_5$. Therefore x_4 is our leaving variable x_l.

2.2 Degeneracy

A BFS is degenerate if one of its basic variables is 0, and can understood through the geometric concept of adjacency. This is because the non-basic variable are forced to 0, meaning we have n tight constraints. However, this assumes that *all* the basic variables are strictly greater than 0 and so when there is a basic variable equal to 0, we have $n + 1$ tight constraints.
This causes issues because, as stated before, adjacency is typically understood as two BFS that share $n - 1$ tight constraints. However, we derived this $n - 1$ term from the assumption that all basic variables are strictly greater than 0. In actuality, it's more like $n - 1 +$ *number of basic variables that can be 0* (see Lecture 3, slide 45 and 47-54 for a geometric visualisation). Note, we see later that this can be more precisely written as $n - 1 +$ *number of surplus variables*, as shown in Section 3.

In the context of the Simplex method, this means that when we pivot between two BFS with a degenerate basic variable, we do not increase the objective function, a phenomenon which we refer to as "stalling". This in itself is annoying in the context for this course and a nightmare for stupid big Linear Programs. But one should know that stalling behaviour can actually lead to a second phenomenon called "cycling", where the Simplex method simply pivots in a loop to the same bases (plural of *basis*) over and over again.

In practice, we can see degeneracy occurs when there are two entering or leaving variables after Gaussian elimination as they "cancel each other out". This is because the variable you didn't pick becomes equal to 0 since that's what happens when you subtract an equally sized

value from something.

In practice we can follow **Bland's rule**:

> *Choose the improving variable with smallest index*, and if there are several possibilities for the *leaving* variable, *also take the one with the smallest index.*

3 Handling Systems in Other Forms

3.1 Converting constraints to something nice

Note that in Section 1, we said that we can make three assumptions and Linear Programs that don't sacrifice generality. There is technically a fourth one: "The RHS of the inequalities must be non-negative." This was excluded because without the right methods (which we are about to cover and you need to know), it does sacrifice generality.
The short of this subsection is summarised below in Equation 2. For reasons understood in Section 6, I will be referring to the right hand side of inequality equations as b, where b_i is the right hand side of the i^{th} constraint.

$$\begin{gathered}\text{Functional Constrain Conversions:}\\ \begin{array}{ccc} x_1 \leq b_i & \rightarrow & x_1 + x_2 = b_i \\ x_1 = b_i & \rightarrow & x_1 + \bar{x}_2 = b_i \\ x_1 \geq b_i,\ \ b_i \geq 0 & \rightarrow & x_1 + -s_2 + \bar{x}_3 = b_i \\ x_1 \geq b_i,\ \ b_i < 0 & \rightarrow & x_i \leq b_i,\ \ b_i \geq 0 \end{array}\\ \text{Non-negativity Constrain Conversions:}\\ \begin{array}{ccc} x_1 \geq RHS_i,\ \ RHS_i < 0 & \rightarrow & x_1' = (x_1 + RHS_i), \quad x_1' \geq 0 \\ x_1 \textit{is unconstrained} & \rightarrow & x_1 = (x_1^+ - x^*), \quad x_1^+, x^* \geq 0 \end{array}\end{gathered} \tag{2}$$

Note that slack and artificial variables are *always* initially basic while decision and surplus variables are *always* initially non-basic. Moreover, when a surplus variable is basic its complimentary artificial variable must be non-basic and vice versa since *the two columns are linearly dependent if they are both basic or both non-basic.*
For similar reasons, x^* will never be lower than the lowest unconstrained variable. Recall to make x^* the lowest unconstrained variable's value to minimise the number of new variable you add to the system.

On the tightness of constraints, recall they are so when the augmentation variables are equal to zero. Just like with *lesser than*, we have to consider the concepts of satisfying and violating the original constraints that are *equal to* and *greater than* in tandem with tightness.

When an artificial variable for an equality constraint is anything but 0, this means that the inequality is not satisfied with equality, and thus the original equality is not satisfied but is in fact violated.

For greater than or equal to constraints it is similar but also a bit more complicated, where we look at the artificial variable minus surplus variable $(\bar{x}_i - s_j)$.

When $(\bar{x}_i - s_j) > 0$, or when $\bar{x}_i > s_j$, the constraint is violated for similar reasons to before: the inequality is not satisfied with equality and so the equality is not satisfied.
Thinking about this graphically helps. When $\bar{x}_i$ is positive and s_j is, say, zero, then $(\bar{x}_i - s_j)$ is positive and indicates that the original decision variable portion x_k of the equality is loose. Thus x_k is too small to reach the boundary and is not satisfying the equality, let alone being above the boundary.

When $(\bar{x}_i - s_j) < 0$, or when $\bar{x}_i < s_j$, it is satisfied but not tight. Again, thinking graphically, if $\bar{x}_i$ is now, say, 0 and s_j is now positive, then $(\bar{x}_i - s_j)$ is negative. It would mean that the decision portion x_k is actually overshooting past the functional constraint boundary and is firmly above it, since s_j still has to cancel out the surplus. This satisfies the original inequality but it is still not tight.

Lastly when $(\bar{x}_i - s_j) = 0$, or when $\bar{x}_i = s_j$ it is both satisfied and tight, for reasons I hope I have made clear.

3.2 The Big-M Method: Intuition and Artificial Variables

When you have an equality in the original system, we introduce an artificial variable $\bar{x}_i$ that allows the equality to be violated. However, because the value of $\bar{x}_i$ represents an *artificial gap* (between the current basic solution and the functional constraint boundary) which **does not exist in the original system**, we want to make sure we don't inadvertently maximise its value. We do this by attaching a massive penalty M to the artificial variable $\bar{x}_i$ in the objective function:

$$\text{Maximise/minimise:} \quad Z = 3x_1 + 5x_2$$

$$\text{subject to:} \quad 3x_1 + 2x_2 = 18,$$

$$x_1, x_2 \geq 0$$

which becomes

$$\text{Maximise/minimise:} \quad Z = 3x_1 + 5x_2 - M\bar{x}_3$$

$$\text{subject to:} \quad 3x_1 + 2x_2 + \bar{x}_3 = 18,$$

$$x_1, x_2, \bar{x}_3 \geq 0.$$

At the start of the Simplex method, non-decision variables are basic and so any artificial variables must have 0 in their top row, but this is a contradiction with what I just showed! (See the $-M\bar{x}_3$ in Z) Therefore, we must perform Gaussian elimination to enforce the 0-1 pattern, which involves moving the M value to other, non-basic top row spots. Note that representing M works better symbolically rather than literally.

After sufficiently many pivots, M should have been removed from the right hand side of the objective function otherwise the Linear Program is infeasible (you *cannot* maximise M). Only after the artificial variables have been removed from the basis, and M has been removed as just stated, has the Linear Program moved into the feasibly region.

Lecture 5, slide 16 (2019) shows a worked example of the Big-M method, where iteration 0 already have the corrective Gaussian elimination done to remove the -M from the objective function. It is the same system as used above to show how to add artificial variables but with all the remaining constraints properly there.

3.3 Two-Phase Method: Application and Surplus Variables

Now, we know about how artificial variables allow artificially mimicking the equalities in the original system, we can now address *greater than or equal to* inequalities.
We handle these with not just an artificial variable but also a surplus variable. The surplus variable acts identically to a slack variable, except instead of "picking up the slack" and filling in between the left and right hand sides of its inequality, it removes the "spilling over surplus value" above the constraint boundary that decision variables generate.
Because we once again generate artificial variables, we need to employ the Big-M method again to handle them properly. However, there is actually a smarter, more refined approach than the Big-M method called *the Two-Phase method*. The first phase involves entering the feasible region by minimising the sum of the artificial variables, where if this fails we can stop. The second involves actually optimising the program.

This method **does NOT** involve an M term in any way. We covered the Big-M method mainly for intuition than application.

Steps for performing the Two-Phase method:

1.1 Convert the original simplex into the augmented system, where you also subtract each artificial variable from the objective function. Do check your plus and minus signs.

1.2 Convert the augmented system into tableau form.

1.3 Change the objective function *without changing any other part of the program* to all zeros including the right hand side *except* the columns belonging to artificial variables. Set those to 1 in the objective function.

1.4 Perform Gaussian elimination to remove the artificial variables from the objective function and firmly placing them in the basis. This will put surplus variables into the top row as positives.

1.5 As with the Simplex method, simply start pivoting as normal until you have removed the artificial variables from the basis. **While this sounds like you're just undoing what you did, trust me you're ~~probably~~ not.**

2.1 Once they are out of the basis, check if the right hand side of the objective function is equal to 0. If not, the Linear Program is infeasible and you can go home.

2.2 Otherwise, create a similar, second tableau by **dropping** the artificial columns of the first tableau. This is to make sure we don't undo our minimisation from phase 1.

2.3 Swap out the objective function with the original objective function of the augmented system. We can do this because it its right hand side value is also equal to 0.

2.4 Simplex away until you've reached optimality!

4 Post-optimality Analysis

This section whisks through three main topics: re-optimisation, shadow prices and sensitivity analysis which are all related to the same practical situation (see current Section title). However, since Sensitivity analysis is a large topic, it has its own Section (see Section 8).

After you've solved a Linear Program (you've found an optimal basic solution), somebody (probably your boss) asks you to solve a very similar Linear Program that is the same as the first except with one or a few coefficients changed. This new Linear Program's solution will not be too far from the one you just found, so we can "hot-start" the solving form the previous optimum.

Re-optimisation
Sometimes it is the case that the values changed has caused the previous optimum to become infeasible! For this, we use *the Dual Simplex method* explained in Section 7.
It is pertinent to note that there are two cases where you re-optimise a solution. The first is when you have lost both feasibility and optimality. The second is when you manage to maintain optimality *but you have lost feasibility*, making the solution something called "super-optimal". This is again explained in Section 7 (specifically Figure 9), where we see that this solution is feasible in the dual form but not optimal.

Shadow Prices
When the values changed in the Linear Program are the right hand side of the functional constraints and they haven't been changed too much as to keep the optimum basis as feasible. If the change is too large then the optimum basis for the second Linear Program "flips" to another. Fortunately we can see what the allowable range of change is through *sensitivity analysis*, which we will touch upon next. It should be noted that shadow prices themselves are a form of sensitivity analysis and can be better understood in that section (specifically Section 8.7).
The utility in this is that *we do not need to execute any iterations of the Simplex method at all*, which is very convenient in real life situations.

Sensitivity Analysis
As stated before, this is explored thoroughly in Section 8. It covers the topics of whether the previous optimum is still feasible, optimal as well as covering allowable ranges of changes to the system while maintaining that optimum. This is useful because we can derive the value of Z and improve on maximising it further through this.

5 A Short Pit-stop in Intuition Town (optional-ish)

So around this point the the course, the flow is actually disrupted a bit. It's logical for us to talk about duality, as it follows from re-optimisation, and then sensitivity analysis, as it follows from shadow prices. But for some reason beyond my puny brain's understanding, it doesn't and we talk about all this other stuff instead.
Because I don't really have the heart nor the balls to rearrange the ordering, I'm just going to have this bizarre chapter... here.
Lecture 7ii, slide 29 onward covers non-examinable topics that I don't care to understand nor write up notes on, despite its compelling sexiness.

If you think you're above this then please go straight to Section 6 for the Matrix form and the Algebraic Simplex Tableau.

5.1 Considerations for the Computer Implementation of Solvers

By understanding the Simplex tableau in it's matrix form (which we explore in Section 6) it allows for using the *revised Simplex method*. This method is used because all the relevant data is stored extraordinarily compactly.
In terms of run time, the number of constraints is a lot more important than the number of variables, which links to the Dual Simplex method. There are also specialised types of Simplex methods for specific types of problems such as the *Transportation Simplex method and the Network Simplex method* (covered in the third-year ORCS course). Moreover, advanced solvers will frequently try to exploit the fact that constraint matrix is very sparse in practice to speed up the process.

5.2 Interior-point Methods for Solving Linear Programs

So we've covered the Simplex method which pivots from corner-point to corner-point to reach an optimum BFS. However, and in a sense obviously, creeping around the edge of the decision boundaries is not the only approach to solving a Linear Program.
Interior-point methods can actually be faster than the Simplex method in certain cases, such as when there is a large amount of corner-points to pivot between.
The reason for Interior-point methods not being frequently used is because while they do typically require less iterations in total, each iteration takes a very long time to calculate. They also can't provide as much auxiliary information about the solution it finds. An

attempt to remedy this is to employ a cross-over algorithm which gets close to an optimum through an interior-point solver, and then finished off with Simplex. However, finding a BFS near one of the interior-points is no easy task!

5.3 The "Line Segment" Theorem and Convexity

The line segments theorem is this; any and all line segments from two points within or on the edge of a feasible region can be expressed as:

$$\Delta x^* + (1 - \Delta)x^{**} \quad \text{where} \quad 0 \leq \Delta \leq 1 \tag{3}$$

which is fancy-math speak for **if you draw a line between *any* two points in the feasible region, then *all* points on that line are also in the set** where the feasible region of a Linear Program is known mathematically as a *convex set*. Note, it is a convex set because the region cannot curve inwards (like a cave), it can only be flat or protrude (see Figure 4 for abstract examples or revisit Figure 1 from Section 2 for a concrete system).

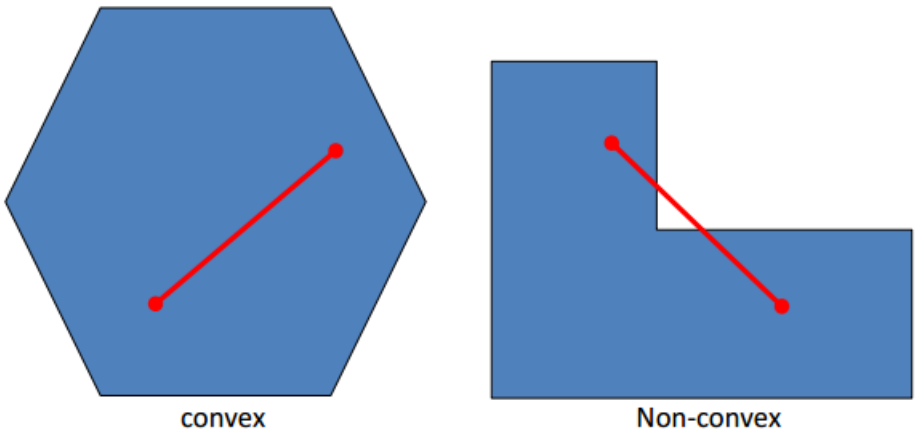

Figure 4: A visual example of the line segment theorem for a convex polytope and concave polytope (Lecture 7ii, slide 4, 2019).

I just used the term "polytope" in the above Figure's caption. This fancy word is just the name of the shape our bounded feasible regions have. Conversely, unbounded feasible regions are polyhedrons.

This concept is useful for proving corner-points are linearly independent since you cannot express one corner-point as the sum of others, otherwise it is not a corner-point! Visualise a corner-point that isn't protruding. Spoilers, you can't do it.
The formal definition for this phenomenon is as follows:

> "A CPF solution is any point x^* in the feasible region that does not lie on the line segment of two feasible points x', x" where neither x' nor x" is equal to x^*."

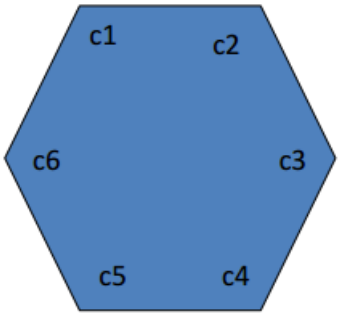

Figure 5: The labelled corners cannot be expressed as the sum of other points in the convex set (Lecture 7ii, slide 8, 2019).

This somehow links back into degeneracy in ways I do not fully understand, let alone am able to articulate.

6 Algebraic Simplex Tableau

We have looked at the Simplex method using the tableau form so far, but now we will consider the ***matrix form*** which allows us to view and generate the values of a tableau conveniently using compactly stored inputs (I will show more, so its okay if this sounds a bit spooky).

TABLE 5.8 Initial and later simplex tableaux in matrix form

Iteration	Basic Variable	Eq.	Coefficient of: Z	Original Variables	Slack Variables	Right Side
0	Z $\mathbf{x}_B$	(0) (1, 2, . . . , m)	1 **0**	$-\mathbf{c}$ **A**	**0** **I**	0 **b**
Any	Z $\mathbf{x}_B$	(0) (1, 2, . . . , m)	1 **0**	$\mathbf{c}_B\mathbf{B}^{-1}\mathbf{A} - \mathbf{c}$ $\mathbf{B}^{-1}\mathbf{A}$	$\mathbf{c}_B\mathbf{B}^{-1}$ $\mathbf{B}^{-1}$	$\mathbf{c}_B\mathbf{B}^{-1}\mathbf{b}$ $\mathbf{B}^{-1}\mathbf{b}$

Figure 6: The magical matrix form of the Simplex tableau, a formulation that will be given to us when necessary (Lecture 8, slide 7, 2019).

The top row with iteration 0 represents the algebraic version of the tableau *from when we construct it using the augmented system, before we've done any pivoting.*
The bottom row represents how we can generate the tableau at *any iteration* i using the corresponding vectors and matrices for that iteration. It shoud be noted that we can already

see that $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{c}$ *do not change* as the tableau pivots.

From this we can deduce that the values $\mathbf{c_B}$ and $\mathbf{B^{-1}}$ do change over time. We can view $\mathbf{B^{-1}}$ as *an encoding of the Simplex row operations performed to move from the original augmented system to the current iteration.* along with $\mathbf{c_B B^{-1}}$ Then straight $\mathbf{c_B}$ is a lot more boring and plainly represents *the current basis' values in the original objective function.* E.g. for the initial BFS of the simplex method, the slack variables, say $\{x_3, x_4, x_5\}$, are the current basis and they (by definition) never appear in the original system's objective function Z, so $\mathbf{c_B}$ at this point is trivially derived as $\{0, 0, 0\}$.

I'd like to emphasis, that ***I* don't *believe* it is *super* important to know what these mean, just *know what to do* when you have them and how to use them to generate corresponding values**, at least in the context of passing the exam ~~please don't be mad Steven~~. I'm saying this so you don't let this scare you more so than anything else. For example, if you know the sub-matrix for the slack variables $\mathbf{B^{-1}}$, then you can easily generate the sub-matrix for the original decision variables (as we are always given the original linear program, so we always know $\mathbf{A}$).
Allegedly, the matrix form is so powerful that we can construct an entire tableau just from knowing the basis.

I hope from all this, the connection between the matrix form and sensitivity analysis is obvious. Because we have an optimal tableau encoded though $\mathbf{c_B}$ and $\mathbf{B^{-1}}$ we can play around with the initial variables $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{c}$ to see allowable increases and decreases in values while maintaining the found solution as the optimum. Naturally, we will explore this thoroughly later, don't worry your pretty little head about it now.
Next, we will discuss the *Revised Simplex method*, where we "project" the pivots of the old $\mathbf{B^{-1}}$ onto a new $\mathbf{B^{-1}}$ which has a complexity $O(m^2)$. This is far less than the complexity of $O(m^3)$ from inverting the $\mathbf{B}$ matrix.
So, to clarify, the revised Simplex method is the same as the original method, except you use a $\mathbf{B^{-1}}$ that solves another linear program and apply it to a new one to quick-start the process. After this you dynamically update it to obtain a $\mathbf{B^{-1}}$ befitting of this new linear program.

7 Duality

Suppose we want to *systematically* determine the best *upper bound* for an objective function. We can take the linear sums of each variable such that it is bigger than its coefficient in the objective function (thus making it an upper bound) but such that the sums of the right hand side are as low as possible. We can see this in the following example:

TABLE 6.1 Primal and dual problems for the Wyndor Glass Co. example

Primal Problem in Algebraic Form

Maximize $Z = 3x_1 + 5x_2$,

subject to

$$x_1 \leq 4$$
$$2x_2 \leq 12$$
$$3x_1 + 2x_2 \leq 18$$

and $x_1 \geq 0, \quad x_2 \geq 0.$

Dual Problem in Algebraic Form

Minimize $W = 4y_1 + 12y_2 + 18y_3$,

subject to

$$y_1 + 3y_3 \geq 3$$
$$2y_2 + 2y_3 \geq 5$$

and $y_1 \geq 0, \quad y_2 \geq 0, \quad y_3 \geq 0.$

Primal Problem in Matrix Form

Maximize $Z = [3, 5]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$,

subject to

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 4 \\ 12 \\ 18 \end{bmatrix}$$

and

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Dual Problem in Matrix Form

Minimize $W = [y_1, y_2, y_3]\begin{bmatrix} 4 \\ 12 \\ 18 \end{bmatrix}$

subject to

$$[y_1, y_2, y_3]\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 3 & 2 \end{bmatrix} \geq [3, 5]$$

and

$$[y_1, y_2, y_3] \geq [0, 0, 0].$$

Figure 7: A system being shown in both its primal (left) and dual (right) forms, being expressed plainly (top) and algebraically (bottom) (Lecture 9i, slide 15, 2019).

So there's a lot going on in this diagram. Let's focus on the top two boxes. The top left box is an example we're familiar with, and we shall call this the *11primal form*". Conversely, the top right box is *"the dual"* of our primal. We construct it by taking the variables in the constraints, say $(1)x_1$, and reorganising them into their own constraints. So the top constraint of the primal is $y_1 + (3)y_3 \geq 3$, which corresponds to the x_1 in the top constraint, the $3x_1$ in the bottom constraint **and the corresponding $\mathbf{x_1}$ value in the objective function 3**. *Hence constructing the dual is a systematic way of creating an upper bound for our primal linear program.*

By following this pattern, we can easily convert between dual and primal forms, where the primal is just the original linear program you are working with and isn't linked to anything tangible (at least not that you need to worry about). This means a primal program *can* be either maximising or minimising. (As per Steven's preference, we will assume the primal is a maximising program.)
You. can stare at the bottom row of Figure 7 if you want but it does not concern me.

We will now recall the generalisation we made for primal linear programs in Section 1 and expand it to cover the dual generalisation where y is a length-m row vector:

$$\text{Minimise/maximise:} \quad W = y\mathbf{b}$$

$$\text{subject to:} \quad y\mathbf{A} \geq \mathbf{c}$$

$$\text{where } y_i, \geq 0, \quad i \, \epsilon \, \mathbb{Z}, \quad i > 0, \quad i \leq m.$$

Now, there is a really cool relationship between the primal and the dual which we will exploit in this course (and even if you don't it's still just really cool). Because the dual is an upper bound on the primal, we can conversely see that *the primal is the lower bound on the dual* since we can construct primal from the dual in the same way we constructed the dual from the prime. And so, we realise that ***the optimal value for the dual program has the exact same value as the optimal value for the primal.*** This is because the constraint boundaries intersect, graphically speaking. This idea is formalised in the *Strong Duality theorem*, defined by *von Neumann* (1944) where:

> "One of the following situations hold:
>
> 1. Both the primal and the dual linear programs are feasible, and for any optimal solution x^* of the primal and y^* of the dual $cx^* = y^*b$.
> 2. The primal is infeasible and the dual is unbounded.
> 3. The primal is unbounded and the dual is infeasible.
> 4. Both linear programs are infeasible."

For my fellow man that shares my contempt for words, let this diagram more eloquently explain the same thing:

		dual F	dual U	dual I
primal	F	YES	NO	NO
	U	NO	NO	YES
	I	NO	YES	YES

Figure 8: The various manifestations of the strong duality theorem. **F** stands for *feasible*, **U** for *unbounded* and **I** for *infeasible* (Lecture 9i, slide 27, 2019).

Please note it is easy to prove all of these except for "if the primal is feasible then the dual is also feasible", which we will cover in Section 7.2.

I mentioned the concept of *super-optimal* back in Section 4, and now I feel the need to deliver. As stated before, super-optimality is when the primal is optimal but infeasible, thus conversely the dual is sub-optimal but feasible.

Here is a *very* important graphic to illustrate how all these optimality types relate to one another:

Figure 9: Nothing in this course matters more than this (Lecture 9i, slide 34, 2019).

I seriously *cannot* ***stress*** the ***importance*** of understanding and resonating with this figure on a ***spiritual*** level. If you are totally on board with it (along with everything else we've covered up until now), then it is likely you will be able to, given some practice, pass the exam - although no doubt by a hairs width. Good job.

7.1 Adapting to Other Primal Forms

Similar to Section 3, we may need to handle a linear program that's in a bit of an ugly form. Thankfully we have an approach for converting our "unconventionally attractice" programs into their dual counterparts without needing to do anything particularly taxing. This is both a cleaner conversion and helps with interpreting our dual more pragmatically.

TABLE 6.14 Corresponding primal-dual forms

Label	Primal Problem (or Dual Problem)	Dual Problem (or Primal Problem)
	Maximize Z (or W)	Minimize W (or Z)
	Constraint i:	Variable y_i (or x_i):
Sensible	$\leq$ form $\longleftrightarrow$	$y_i \geq 0$
Odd	$=$ form $\longleftrightarrow$	Unconstrained
Bizarre	$\geq$ form $\longleftrightarrow$	$y_i' \leq 0$
	Variable x_j (or y_j):	Constraint j:
Sensible	$x_j \geq 0$ $\longleftrightarrow$	$\geq$ form
Odd	Unconstrained $\longleftrightarrow$	$=$ form
Bizarre	$x_j' \leq 0$ $\longleftrightarrow$	$\leq$ form

Figure 10: The table for Strange, Odd and Bizarre primal-dual conversions. Lovingly referred to by many as the *"Son of a Bitch"* conversions, or more colloquially spoken as *"Sonnovabitch"* (Lecture 9ii, slide 14, 2019).

These conversions are equal to if one converted the primal program to standard form and then created the dual from that. But its nice to have when you're feeling lazy.

7.2 Duality and the Algebraic Simplex Tableau

Let me emphasis duality exists independently from the Simplex method (or any method for solving linear programs for that matter). Duality just is. However, this doesn't stop these two from being elegantly interlinked with one another.

Recall how $\mathbf{c_B B^{-1}}$ was a crucial encoding of the elementary row operations that delivers the original linear program to a given iteration in one step. If we re-contextualise $\mathbf{c_B B^{-1}}$ as the y featured in the generalised primal form and re-arrange an equation:

$$\begin{aligned} \text{Minimise/maximise:} \quad & W = \underline{y\mathbf{b}} \\ \text{subject to:} \quad & y\mathbf{A} \geq \mathbf{c} \rightarrow \quad \underline{y\mathbf{A} - \mathbf{c}} \geq 0 \\ & \text{where } \underline{y} \geq 0 \end{aligned}$$

we find that all the underlined terms actually appear in top row of the primal tableau:

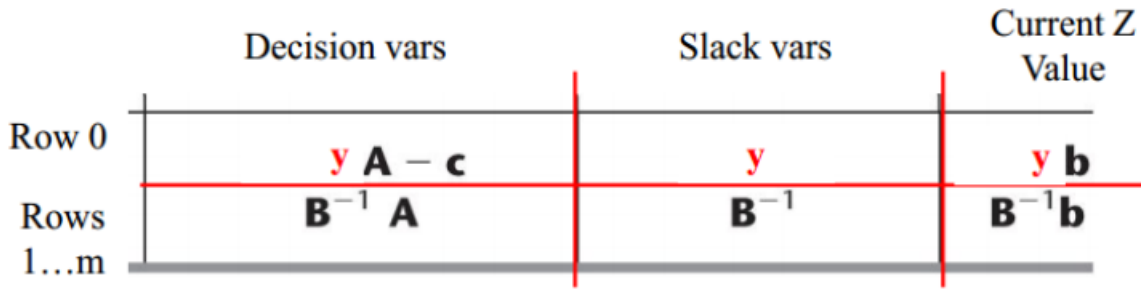

Figure 11: For in case my words have failed you, as I expect they have. $\mathbf{c_B B^{-1}}$ has undergone the y substitution realised (Lecture 10, slide 7, 2019).

The top row of the tableau always encodes a *complimentary* dual solution which maintains the same value such that $Z = W$. However, usually this complementary dual solution will *not* be feasible.

The coefficients of the primal decision variables correspond not to the dual decision variables *but* the dual surplus variables. Vice versa is also true. Do note that when a primal variable x_i is basic its dual equivalent y_i is non-basic and vice versa once again.
Now is a good time to recall our optimality condition of "the top row must be non-negative" and see that it corresponds to the dual decision and surplus variables being non-negative, which is synonymous with the dual being feasible!

So, in a sense, the Simplex method attempts to obtain dual feasibility while not losing primal feasibility. And moreover, the Simplex encodes an optimum solution for the primal and a complimentary optimum solution for the dual.

Now, to return to proving Strong Duality which we discussed in Figure 8. We do this is seven steps:

1. Assume, without loss of generality, that the primal is both *feasible* and it has *a finite optimum*.
2. If we run the Simplex method with Bland's rule, it is guaranteed to terminate in a finite number of steps; the top row will eventually be all non-negative.
3. Let Z^* be the value of the primal objective function found upon termination.
4. This top row equal to Z^* encodes a dual solution that is feasible for the dual and achieves $Z^* = W^*$
5. By weak duality, Z^* is the best for the primal and W^* is the best for the dual.
6. To emphasise, the dual is feasible with a finite optimum which is equal to the primal.
7. Corollary, the "top row non-negative" condition for termination is correct since it guarantees optimality.

Note how we did *not* need to prove the Simplex was optimal, only that it terminates in finite time. Instead, the correctness of the Simplex is proved corollary (follows on through) from the rest of the proof.

Let us also note graphically the connection between the primal and dual corner-points. As the Simplex pivots from primal corner-point to corner-point, it does so too for the dual. However, the only basic feasible solutions of the dual exist when they intersect at corner-points from the primal program, since they are feasible there.

8 Sensitivity Analysis

Recall the algebraic table from Section 6 and recognise it in this figure:

TABLE 6.17 Revised final simplex tableau resulting from changes in original model

		Coefficient of:			
	Eq.	Z	Original Variables	Slack Variables	Right Side
New initial tableau	(0)	1	$-\bar{\mathbf{c}}$	**0**	0
	(1, 2, . . . , m)	**0**	$\bar{\mathbf{A}}$	**I**	$\bar{\mathbf{b}}$
Revised final tableau	(0)	1	$\mathbf{z}^* - \bar{\mathbf{c}} = \mathbf{y}^*\bar{\mathbf{A}} - \bar{\mathbf{c}}$	$\mathbf{y}^*$	$Z^* = \mathbf{y}^*\bar{\mathbf{b}}$
	(1, 2, . . . , m)	**0**	$\mathbf{A}^* = \mathbf{S}^*\bar{\mathbf{A}}$	$\mathbf{S}^*$	$\mathbf{b}^* = \mathbf{S}^*\bar{\mathbf{b}}$

Figure 12: Modified algebraic for, "simplified" for sensitivity analysis (Lecture 13, slide 47, 2019).

Now, you may be wondering why have decided to bastardise the original algebra into... that. The reason is because we will be manipulating it extensively to perform sensitivity analysis, seeing how much certain values can change while maintaining optimality. So this is a lot cleaner for writing out equations, gets you in the right head space and helps avoids any silly mistakes like missing a term.

8.1 Changing a Non-basic Variable

You've solved a linear program but your boss now want you to solve the exact same linear program except for some differing coefficients of the decision variable x_k which is non-basic in your old optimal solution. This means that we need to change A, c or both in the column corresponding to x_k.

Is the old solution still feasible? Yes!
This is because we are not changing the b vector, thus the values of the basic variables remain non-negative and feasible.
Is the old solution still optimal? Maybe!
To check, get the dual constraint for x_k and see if it is violated or not. You can do this since you are given the final, optimal tableau for the primal. Recall the values of the dual decision variables is the top row of the primal slack variables $y*$.

8.2 Adding a Non-basic (decision) Variable

You've solved a linear program but your boss now want you to solve the exact same linear program except there is now an additional decision variable x_{new}. We do this easily because, due to it being non-basic and thus equal to 0, we can assume it was implicitly always there in the original linear program.

Is the old solution still feasible? Yes!
This is because even though you are adding a new variable to each constraint, since it is a non-basic variable, they all evaluate to 0 and thus the left hand side value of the constraints remain unchanged. Moreover, the b portion of the table isn't being changed, so the basic variables remain non-negative an feasible.
Is the old solution still optimal? Maybe!
Because the new variable is a decision variable, the slack remains unchanged which recall correspond to the dual solution. This means that the dual solution remains unchanged, and thus is still feasible while the primal is feasible *except* for the new constraint we just added to the dual (because you make a new dual constraint when you add a primal decision variable). So all we need to check is whether the old dual solution satisfies this new constraint, where the values of the dual decision variables is the top row of the primal slack variables $y*$.

8.3 Finding the Allowable Change of a Non-basic Variable

You've solved a linear program but your boss now want you to solve the exact same linear program except for the c vector where some non-basic variables have had their coefficients modified. (See Section 8.7 for a similar example partially worked.)

Is the old solution still feasible? Yes!
This is because we are not changing the b vector, thus the values of the basic variables remain non-negative and feasible.
Is the old solution still optimal? Maybe!
As long as the corresponding dual constraints are not violated, then it is optimal. Therefore, we can find the range of increase Δc, which does not need to be bounded on both sides, for every c value that we changed.

8.4 Changing a Basic Variable

You've solved a linear program but your boss now want you to solve the exact same linear program except that a column of a basic variable x_b has been changed.

Is the old solution still feasible? Maybe! What about optimality? Still maybe!?
Now, changing this comes with an annoying caveat that you may need to perform Gaussian correction after changing the values in order to restore the 0-1 pattern, and until then we cannot know whether the program is feasible, optimal, both or neither.

8.5 Finding the Allowable Change of a Basic Variable

You've solved a linear program but your boss now want you to solve the exact same linear program except now c has a value changed it in, and it belongs to a basic variable in the optimal solution. (See Section 8.7 for a similar example partially worked.)

Is the old solution still feasible? Yes!
This is because we are not changing the c vector, thus the values of the basic variables remain non-negative and feasible.
Is the old solution still optimal? Maybe, we need to correct it first!
Let us consider the formula

$$y^* \bar{A}_i - \bar{c}_i \quad = \quad y^* \bar{A}_i - (c + \Delta c_i) \quad = \quad z_i^* - \Delta c_i$$

where we must consider that the top row value z_i^* was ultimately 0 so changing the initial c_i for a basic variable makes the new top row value $-\Delta c_i$. Therefore we must perform Gaussian elimination before we can discern anything more.
Once you have, you will have generated the new top row of the new program, including $y*$. Firstly, from this you can generate a set of inequalities because some terms will inevitably include Δc_i (you had to subtract it from the top row and enforce the 0-1 pattern). For each term which is non-basic and contains Δc_i, we must make sure they are strictly positive. From this requirement the inequalities are clearly generated. And from this, we can find the allowable range of increase and decrease for Δc_i as well as the range for c_i itself.
Secondly, the aforementioned y^* that we generated also means that we can compute the right hand side of the objective function (recall the algebraic form). And since we just computed the upper bound on Δc_i, we can compute the maximal value for the current basis as well.

8.6 Adding a New Constraint

You've solved a linear program but your boss now want you to solve the exact same linear program except now they want you to toss in an extra constraint to mix things up. Great. Graphically, please recall adding a constraint means that we now have a smaller feasible region.
Note that implementing this usually involves the same old same old. Create a corresponding slack variable for the constraint, add the initial row of the augmented system to the final form of the tableau. Re-optimise with Gaussian elimination to correct if necessary.

Is the old solution still feasible? Maybe!
Simply evaluate the new constraint with the current optimal solution. If it's satisfied, hurrah!
Is the old solution still optimal? Yes!
Yes, because we have not changed any part of the tableau that concerns itself with optimality, only restricted the feasible region. Meaning super-optimality is likely and re-optimisation via the dual Simplex method may be necessary.

8.7 Changing a Right Hand Side Value: Eduardo's Changes I

You've solved a linear program but your boss now want you to solve the exact same linear program except for the b vector, specifically the value in the i^{th} row, b_i. To give a concrete example:

$$b = \begin{bmatrix} 4 \\ 12 \\ 18 \end{bmatrix} \rightarrow \bar{b} = \begin{bmatrix} 4 \\ 24 \\ 18 \end{bmatrix}$$

where $\bar{b}$ is a modified version of the original b vector and we only change the second row, corresponding to the second augmentation variable and its column.

Is the old solution still feasible? Maybe!
We can see from Figure 12 that changing b involves changing all the right hand side values, so we are playing around with not just optimality but feasibility here.
We can figure out whether feasibility is maintained through incremental analysis.
Observe that $\bar{b} = b + \Delta b$, and so we can deduce $S^*\bar{b} = S^*b + (S^*)\Delta b$. So we can understand our concrete example as:

$$\bar{b} \;=\; b + \Delta b \;=\; \begin{bmatrix} 2 \\ 6 \\ 2 \end{bmatrix} + \begin{bmatrix} 1/2 \\ 1/3 \\ -1/3 \end{bmatrix} \Delta b_2$$

where the values $[1/2,\ 1/3,\ 1/3]$ come from the corresponding second column of matrix S^* in the example, given we only changed the second row of b. Looking at it in terms of matrix multiplication makes it clear why this is the case. However, please recall that these new values *must* be non-negative, so we can rearrange the equations like so:

$$\begin{array}{lcrcc} 0 \leq 2 + (1/2)\Delta b_2 & \rightarrow & -6 & \leq & \Delta b_2 \\ 0 \leq 2 + (1/2)\Delta b_2 & \rightarrow & -6 & \leq & \Delta b_2 \\ 0 \leq 2 - (1/3)\Delta b_2 & \rightarrow & \Delta b_2 & \leq & 6 \end{array}$$

which can all be written into:

$$-12 \leq -6 \leq \Delta \leq 6$$

Thus we can plainly see that there is an allowable increase of 6 units and an allowable decrease of 6 unites for b_2. We can extrapolate this into a concrete limit as:

$$0 \leq b_2 \leq 12$$

This is the same range as previously stated, but with the original b_2 value of 6 added to it. Changing b_2 outside of this range leaves us with no guarantees of feasibility nor optimality.

Is the old solution still optimal? Yes!
We only play around with the right hand side of things, which surprisingly means that we do not actually change anything related to the optimality condition itself. This means changing b will keep you old solution as optimal or it will make it super-optimal in the new system, at which point re-optimisation through the dual Simplex method is needed.

I want my Shadow Price explanation...
Ah yes, this needlessly confusing topic. Essentially, the top-row values of the augmented variables which correspond to the m dual decision variables have an analytical meaning; for every unit you increase b_i, the objective function's value increases by y_i. That's it.
Just to reiterate, the shadow price for each constraint is the constraint's respective z_i^* value in the optimal tableau.

8.8 Changing All Right Hand Side Values: Eduardo's Changes II

You've solved a linear program but your boss now want you to solve the exact same linear program except the entire b vector has been changed.

Is the old solution still feasible? Maybe!
The right hand side variables must be non-negative, so we can loose feasibility. You can derive the ranges in a similar way. However, if you are feeling strict and have already done incremental analysis where we change one value of b, you can utilise the 100% rule, a pessimistic test for the allowable ranged.

In the previous example we found a range of 12 between out allowable increase and decrease. Meaning if we take that as an upper bound, as long as *the sum of the changes* we make to each b value does not exceed 12 then we will stay in optimality (see Lecture 13, slides 42 and 43 for more information).
Is the old solution still optimal? Yes!
Like in the previous subsection, we are only changing the right hand side of equations so the optimality condition is never affected, meaning we produce something either optimal or super-optimal. If the latter, we re-optimise through the dual Simplex method.