Class 9 Notes: Introduction to Optimization

Myles D. Garvey, Ph.D

Summer, 2019

Contents

1 Introduction

We are now on one of the most important lectures of our course: optimization. In the previous lectures, we touched lightly on some ways to optimize functions. However, we only had done so in an *unconstrained* and *continuous* manner. Some applications of optimization, however, differ based on the particular situation at hand. For example, if I am trying to find the optimal truck

route between two cities, this is an example of a *discrete optimization* problem, which means that the only allowable values to be assigned to the variables is a countable set. We of course cannot conduct our same tools of calculus (at least in their current form) on countable sets of solutions.

Furthermore, we only looked at problems that were unconstrained. In otherwords, even if our decision variables were continuous, we still allowed the entire Euclidean n dimensional space to be considered for candidate solutions. In practice, managers often face constraints on the decisions they make. Hence, we need to reflect these constraints into our modelling approach by defining reasonable sets of points by way of specifying specific types of relations (equalities and inequalities) that must hold true about our decision variables.

In this lecture, we will review through the fundamentals of optimization. This is a very large field of mathematics and business alike (mainly within Supply Chain Management, Logistics, Operations Research, Marketing, and Finance). We will first review through the different types of optimization. This overview is far from enumerated, and there are many other types. However, I have taken a proactive approach to try and list the most common forms of optimization problems that you are likely to encounter. We will then discuss constrained non-linear optimization, and how we can incorporate these "real-world constraints" into our problem, as well as solve them under certain conditions. After, we will discuss the fundamentals of linear programming, which is a method of solving optimization problems where all equations can be formulated as a continuous linear equation.

The following section will discuss stochastic, deterministic, closed, and dynamic programming approaches. We do not offer a complete review of these, but merely their definition with simple examples. We last will discuss an important characteristic of optimization: problem complexity. We can formulate problems all we would like, but an important consideration that the analyst must keep in mind is, can it be solved? If not, how can we at the very worst approximately solve it? The theory of computational complexity is leveraged to answer this question, and analysts are often concerned with the type of problem they are solving. Knowing the problems *complexity* helps guide the analyst towards the approach they should use to solve it, which is our topic of discussion in the next lecture. For now, let's get to the fundamentals of optimization!

2 Types of Optimization

2.1 Mathematical Programming: A Way to Organize Optimization Problems

While there are many different types of optimization problems, almost all of them are expressed an organized in the same manner. The way in which we organize our optimization problems is through the use of a technique called *mathematical programming*. The elements of the mathematical program are almost always:

1. The input variables for which much be chosen.
2. A collection of equalities and inequalities, referred to as constraints, that help define the set of potential solutions to consider.
3. The objective function, which measures how "good" or "bad" a solution is, and is the function for which must be optimized by considering *feasible solutions* as mandated by the constraints.

For example, suppose we wanted to find the amount of products to order, Q, so as to maximize the overall profit, determined by $\pi(Q) = pQ - c - bQ^2$. Further suppose that the minimum amount that a buyer's contract requires them to order is k units, but the supplier also has capacity constraints, which is indicated to the buyer. Hence, the buyer can only order as high as M units. While this problem is a simple one, we can still organize the entire problem as follows:

$$\begin{aligned}
&\max \pi(Q) = pQ - c - bQ^2 \\
&\text{s.t.} \\
&Q \leq M \\
&Q \geq k \\
&Q \in \mathbb{R}
\end{aligned}$$

The result is called a *mathematical program*. It essentially says to find a value for Q in the set $A = \{x | x \leq M \text{ and } x \geq k\}$ such that $\forall x \in A$, $\pi(Q) \geq \pi(x)$. Almost all optimization problems (with the exception of perhaps some dynamic programming problems) are formulated in this manner. Generally, for any optimization problem, regardless of it's type, we can write:

$$\begin{aligned}
&\max/\min f(x_1, \ldots, x_n) \\
&\text{s.t.} \\
&g_1(x_1, \ldots, x_n) \leq 0 \\
&g_2(x_1, \ldots, x_n) \leq 0 \\
&\vdots \\
&g_m(x_1, \ldots, x_n) \leq 0 \\
&h_1(x_1, \ldots, x_n) = 0 \\
&h_2(x_1, \ldots, x_n) = 0 \\
&\vdots \\
&h_k(x_1, \ldots, x_n) = 0
\end{aligned}$$

The variables x_i are referred to as the *decision variables* and the function $f(x_1, \ldots, x_n)$ is referred to as the *objective function*. The functions $g_i(x_1, \ldots, x_n)$ are referred to as *inequality constraints*. Any inequality can be written in the form of $g_i(x_1, \ldots, x_n) \leq 0$. For example, the inequality $5x_1 - 2x_2^2 + 3 > 5$ can be forced into the above format by first introducing a new variable (often called a *slack* variable) s and rewriting it as: $g(x_1, x_2, s) = -5x_1 + 2x_2^2 + 2 + s$. We can clearly see that writing $5x_1 - 2x_2^2 + 3 > 5$ is equivalent to writing $-5x_1 + 2x_2^2 + 2 + s \leq 0$, or, $g(x_1, x_2, s) \leq 0$. The equations $h_i(x_1, \ldots, x_n) = 0$ are called *equality constraints*, and again, any equality can be expressed in this manner. For example, the constraint $5x_1 - 3x_2 + 4x_3 = 5$ can be expressed as $5x_1 - 3x_2 + 4x_3 - 5 = 0$, and so we would have $h_i(x_1, x_2, x_3) = 5x_1 - 3x_2 + 4x_3 - 5$.

We commonly express optimization problems in general form, like we did above. In the previous example, we had the decision variable Q and the constants M, k, p, c, b. When we assign

actual values to these variable constants, the problem is said to be *instantiated*. We commonly first try to formulate the general problem, and then, when given specific values for the constants, we plug them into the model. For example, if we had $M = 100, k = 10, p = 50, c = 5, b = 2$, our instantiated version of our model would look like:

$$\begin{aligned}
&\max \pi(Q) = 50Q - 5 - 2Q^2 \\
&\text{s.t.} \\
&Q \leq 100 \\
&Q \geq 10 \\
&Q \in \mathbb{R}
\end{aligned}$$

This form is then the form that will be solved, which we will briefly discuss in a later section. The take away here for mathematical programming is that it is a framework that we use to express optimization problems. The most important step in solving any optimization problem is designing the mathematical program in such a way that reflects the intended theoretical environment that is to be modeled. As we progress in the various sections, you will be exposed to how this process generally works. However, since the modelling process is inherently subjective, there is no single "one way" to model. What can be said, however, is that most models will be characterized by the mathematical program. There are many types of programs, and we will explore some of these today. However, here is a brief enumeration of some (but not all) of the different types of mathematical programs:

1. Linear Program - All equations in the objective function and constraints are linear equations. All variables are continuous.
2. Binary Program - All variables can only have a value of 0 or 1. A sub-type of problem is a binary linear program, where all variables are binary and equations are linear.
3. Integer Program - All variables can only have an integer value. A sub-type of problem is a integer linear program, where all variables are integers and all equations are linear.
4. Mixed-Integer Program - Some variables are integers, others are continuous. A sub-type of problem is a mixed-integer linear program, where all equations are linear, some variables are integer, and some variables are continuous.
5. Stochastic Program - Some variables are random variables or some constants are random variables. As such, the use of probability distributions and statistical moments are used in the program. Simulation is a common technique used to solve these types of programs.
6. Dynamic Program - A problem that can be broken into smaller sub-problems, and solve iteratively or recursively.
7. Fuzzy Program - A model that leverages *Fuzzy Logic* and *Fuzzy Sets*. This is beyond outside the scope of our course, but is worth a mention here.
8. Quadratic Program - A model that has quadratic equations in it.

9. Multi-Objective Program - A model that seeks to optimize more than one objective. For example, you may want to maximize profit and simultaneous minimize cost, or you may want to minimize total distance traveled while minimizing total cost.

2.2 Constrained vs. Unconstrained

The first big distinction between types of optimization problems are those problems for which are *constrained* and those for which are *unconstrained*. We have already reviewed through some of the fundamentals of unconstrained optimization. The process for solving these problems entails computing the gradient of functions, setting them equal to zero, and solving. We say that it is unconstrained because we are not putting any restriction on the type of solutions that we are looking for. While unconstrained optimization problems are often easier to approach, since they mainly entail solving systems of equations, this is not always the case. For example, suppose we had the function $f(x) = xe^{-x} - 2x$. If we were to optimize this, we would find the first and second order conditions:

$$\frac{d}{dx}[xe^{-x} - 2x] = e^{-x} - xe^{-x} - 2 = 0$$

As we can see, it seems impossible to solve explicitly for x (try to solve it). We can try to take the log, but this would convert the x to a log of x, and we would be back to the same problem of having x and a transcendental function of x. In other words, it seems impossible to express x by itself on it's own side. In such situations, we don't solve for x, but merely state the equation that must be true about the optimal x. When we do this, then it is said that we are *characterizing the optimal solution*. Sometimes, we cannot find an explicit optimal solution for unconstrained optimization problems. We instead need to resort to characterization by looking at relationships, as well as upper and lower bounds of the optimal solution. Let us first look at the second order conditions for a maximum:

$$\begin{aligned}
&\frac{d^2f}{dx^2} = \frac{d}{dx}[e^{-x} - xe^{-x} - 2] < 0 \\
&-e^{-x} - e^{-x} + xe^{-x} < 0 \\
&-1 - 1 + x < 0 \\
&x < 2
\end{aligned}$$

So we know that if we are to have a maximum value at a value for x, this value needs to be less than 2. This leaves us with two possibilities. Either $x \in [0, 2)$, or $x \in (-\infty, 0)$. Let us look at the first possibility. If $x \in [0, 2)$, then $x = 0$ or $x \in (0, 2)$. If $x = 0$. then $f'(0) = -2$. Hence, the maximum cannot be here, since $f'(0) \neq 0$. This means it could be in $(0, 2)$. If $x > 0$, which for our case it is, we notice that $e^{-x} > 0$. Looking at the derivative, we notice that we can express it as $e^{-x}(1 - x) - 2$. In order for $(1 - x) > 0$, we need $x < 1$. We now have two additional cases to observe. Either $x < 1$ or $x > 1$. When $x = 1$, we see that the derivative is -2, so this also cannot be the maximum or minimum. When $x > 1$, $(1 - x) < 0$. Hence, $e^{-x}(x - 1) < 0$. This means that the entire derivative $e^{-x}(x - 1) - 2 < 0$ for all $x > 1$. Hence, we have eliminated all values of $x > 1$ for the possible maximum and minimum, since it is impossible for the derivative to be 0

on this interval. But what about the interval $(0,1)$? We then would have $e^{-x}(1-x)$ as a positive number. However, we notice that for $x \in (0,1)$, $e^{-x} < 1$. Hence, we have $e^{-x}(1-x) < (1-x)$ (since $(1-x) > 0$ on this interval). But when $x \in (0,1)$, we clearly see that $(1-x) < 2$. Hence, we now know that on the interval $(0,1)$, that $e^{-x}(1-x) < 2$, or that $e^{-x}(1-x) - 2 < 0$. Notice that the left hand side of the inequality is our derivative. Hence, we have proven that it is impossible for the derivative to be 0 on the interval $(0,1)$. Overall, we have shown that it is impossible for the maximum to be anywhere in $[0,\infty)$, since the derivative is negative for all values of $x \geq 0$.

As an upper bound, we now know that $x < 0$. Can we find a lower bound? First, notice that the first order conditions can be rewritten as:

$$e^{-x}(x-1) - 2 = 0 \rightarrow x = 1 - \frac{2}{e^{-x}}$$

Now, we know that $x < 0$, which means that $e^{-x} > 1$ Hence:

$$\begin{aligned} e^{-x} &> 1 \\ 1 &> \frac{1}{e^{-x}} \\ 2 &> \frac{2}{e^{-x}} \\ -2 &< -\frac{2}{e^{-x}} \end{aligned}$$

Hence, using this and the first order conditions, we have:

$$x = 1 - \frac{2}{e^{-x}} > 1 - 2 = -1$$

Therefore, we have shown that the maximum needs to be in the interval $[-1,0)$. At $x = -1$, we see that the derivative is $-2e - 2 < 0$. Therefore, we can clearly see that the optimal solution is in the interval $(-1,0)$.

What this illustrates is that even if our optimization problem is unconstrained, this does not necessarily mean it is "easy" to solve through our typical method of "find derivative, set equal to 0, etc". Sometimes, we are left in a position where we cannot explicitly solve for an x, and instead, we need to resort to either (1) approximate optimization (to be discussed in class next session) or (2) characterizing the optimal solution via upper and lower bounds, as well as looking at other criteria that must hold true in order for there to exist an optimal solution. In the example above, we clearly saw that solving for an x explicitly is pretty much impossible (I haven't tried further than this, but perhaps you can!). These instances would then necessitate a careful examination of the behavior of the derivative/gradient in certain neighborhoods of solutions. We then can carefully refine the bounds of our optimal solution by thorough examination of the derivative and second derivative (gradient/Hessian).

2.3 Linear vs. Non-linear

There is yet another common type of optimization problem that have many practical applications. This type is known as *linear optimization*. As the name suggests, linear optimization entails the

use of linear equations and inequalities. If the optimization is unconstrained, linear optimization is easy: the answer is negative infinity or positive infinity (needless to say, very little practical relevance here!). When the optimization is constrained, then this is an entire different ballpark. Linear optimization with constraints is often referred to as *linear programming*. We say that a mathematical program is a linear program if the objective function and the constraints are all linear equations. Recall the definition of a linear equation:

Definition 1 *A linear equation in n variables is any equation that has the form:*

$$a_0 + a_1x_1 + a_2x_2 + \cdots + a_nx_n$$

where $a_i \in \mathbb{R}$

Consider the following problem. Assume that a manufacturer produces two types of products: product 1 and product 2. They would like to know the quantity of how much to produce of each product, represented by the decision variables x_1 and x_2, respectively. Each product yields a unit profit of p_1 and p_2, respectively. However, each product uses a certain number of parts, some of which are shared among the two products. More importantly, the manufacturer has only a fixed quantity of parts in it's inventory. This not makes the problem much more difficult to model, but not impossible!

First, let us consider the objective function. One way to "score" the decision made by the decision maker (which in this instance, entails two decisions: how much to product of product 1 (x_1), and how much to make of product 2 (x_2). We know that the manufacturer can earn p_1 dollars per unit sold of product 1, and they can earn p_2 dollars per unit sold of product 2. If they sell x_1 units of product 1, their total profit from this is p_1x_1. If they sell x_2 units of product 2, their total profit from this is p_2x_2. Therefore, the total profit of the entire decision would be $p_1x_1 + p_2x_2$. We notice that this is a linear equation according to the definition (since it is assumed that p_1 and p_2 are constants).

Now, suppose that product k consumes $a_{k,i}$ units of part i, which has a total inventory of b_i. We need to determine how the decision variables are *constrained* from the problem description. That is, given our problem, we know the manufacturer only has limited quantity of parts. Intuitively, this would mean they have a capacity in how many units x_1 and x_2 they can make. In order to reflect this real world constraint, we need to think about how many units of part k in total are consumed as a result of manufacturing x_1 units of product 1 and x_2 units of product 2. Let us assume that there are a total of 5 different parts. When we manufacture x_1 units of product 1, it will consume $a_{1,1}$ units of part 1 for every unit of product 1 produced. Hence, the amount of units of part 1 that are consumed for producing x_1 units of the product 1 would be $a_{1,1}x_1$. Likewise, the total number of units of part 1 consumed for producing x_2 units of product 2 would be $a_{1,2}x_2$. So, the total number of units of part 1 that would be consumed by the manager's decision would be $a_{1,1}x_1 + a_{1,2}x_2$. But, we know that we only have b_1 units in inventory for the first part. This would mean that the total number of parts consumed must be less than or equal to b_1. In otherwords, we must have $a_{1,1}x_1 + a_{1,2}x_2 \leq b_1$. Following a similar logic for the other parts, we know that generally, we must have $a_{i,1}x_1 + a_{i,2}x_2 \leq b_i \forall i \in \{1, 2, 3, 4, 5\}$. Since this is profit, we ideally want to be maximizing. As such, we can summarize all of our observations here by way of writing the following *linear program*:

$$
\begin{aligned}
&\max\ p_1 x_1 + p_2 x_2 \\
&\text{s.t.} \\
&a_{i,1} x_1 + a_{i,2} x_2 \leq b_i, \forall i \in \{1,2,3,4,5\}
\end{aligned}
$$

or, more explicitly:

$$
\begin{aligned}
&\max\ p_1 x_1 + p_2 x_2 \\
&\text{s.t.} \\
&a_{1,1} x_1 + a_{1,2} x_2 \leq b_1 \\
&a_{2,1} x_1 + a_{2,2} x_2 \leq b_2 \\
&a_{3,1} x_1 + a_{3,2} x_2 \leq b_3 \\
&a_{4,1} x_1 + a_{4,2} x_2 \leq b_4 \\
&a_{5,1} x_1 + a_{5,2} x_2 \leq b_5
\end{aligned}
$$

Notice that this is a linear program because all of the constraints and the objective function are linear equations. We will discuss a method, known as the *Simplex Method*, that is often entailed in solving these types of problems.

2.4 Stochastic vs. Deterministic

There is yet another distinction in types of optimization problems, known as Stochastic and Deterministic programming. At times, we want to model certain decisions or elements in our environment as *random*. That is, we may know the values that could be assigned to certain variables, and the frequency at which they are assigned to certain values, but we may not know the exact values that will be assigned. In such instances, we model these variables as *random variables*. You will cover random variables more thoroughly in your statistics course, so we will not recap them here. However, we will review through an example.

Suppose that in a single week, a retailer would like to know how many units to order so as to minimize their overall costs for the week. The problem is, they do not know their demand, and so we model demand as a random variable $D \sim f(\boldsymbol{x}, \boldsymbol{\theta})$, with cumulative distribution function $F(x, \boldsymbol{\theta})$. Suppose that if the number of units ordered Q exceeds demand D, that the retailer is incurred an inventory holding cost of h dollars per unit (which would be a total cost of $h(Q - D)$. In the opposite case, if the retailer orders too little, they incur an opporunity cost of $(p - c)(D - Q)$. This is the *newsvendor model* that we discussed in an earlier lecture, and hence would have the objective function of $(p - c)\max\{D - Q, 0\} + h\max\{Q - D, 0\}$. The problem, as we encountered in this example, is that D is random, and hence we cannot analyze it in the traditional manner. We can, however, analyze it's *expected value* $E[(p - c)\max\{D - Q, 0\} + h\max\{Q - D, 0\}]$. However, suppose that the retailer would like to maintain a specific service level. That is, they would like to not only order so as to minimize their cost, but also order so as to maintain a minimum

probability α of being in stock $F(Q)$. The mathematical program would hence look like:

$$\min E[(p-c)\max\{D-Q,0\} + h\max\{Q-D,0\}]$$
$$\text{s.t.}$$
$$Q \geq 0$$
$$F(Q) = P(D < Q) \geq \alpha$$

When a mathematical program comprises of any expected values, variances, statistical moments, or probability distributions, then it is called a stochastic program. If a model only contains variables that does not involve random variables or probability, then the model is referred to as a *deterministic model*. In our course, we only consider deterministic models. However, this is not to underscore the usefulness of stochastic models.

2.5 Closed vs. Dynamic Programming

Recall that when we discussed functions and types of functions, there were two ways to specify them: closed form and recursive form. Since functions can be expressed in either manner, we can conduct optimization in either form. When optimization is performed on an objective function that is recursively defined, or, in a manner where we can break the problem into smaller problems for which the solutions can be used to solve other smaller problems, the specific mathematical program is referred to as a dynamic program.

The way to think about dynamic programming is by breaking a larger and more complex problem into smaller sub-problems, and finding the optimal solution to the entire problem by finding the optimal solution to smaller sub-problems. To illustrate this process, we review through a well-known problem called the *shortest path problem*. For example, suppose we have 10 cities, and there are well established connections between the cities. Each connection costs a certain amount of money to travel from one city to another city. The goal is to find the shortest path from an origin city to a destination city. This problem can be solved using the concept of Dynamic Programming, by breaking the overall problem into a series of smaller problems. The algorithm that accomplishes this is called Dijsktra's Algorithm:

```
Let V be a set that will store cities which were visited.  Set it equal to the
    empty set
Let D[i] be the current weight on city i.  Initially, all except the origin
city is set equal to a value of infinity.  The origin is set to 0.
Let D[i,j] be the distance or cost from city i to city j
Let C represent the current city, set this equal to the origin.

While V is not equal to the entire set of cities:
    Get all neighboring cities of city C
    For each neighbor i that has NOT been visited:
        D[i] = min(D[C]+D[i,C],D[i])
    Put C in V
    Set C equal to the city with the lowest D[i] that is not in V
```

How are we breaking this into smaller problems? Notice that the optimization is not occurring on the entire set of cities, but rather, on only a handful of cities. When you look inside the while loop, you will notice a sub-optimization problem being solved. We will illustrate this with a specific example. Suppose we have 7 cities with the following costs between the possible

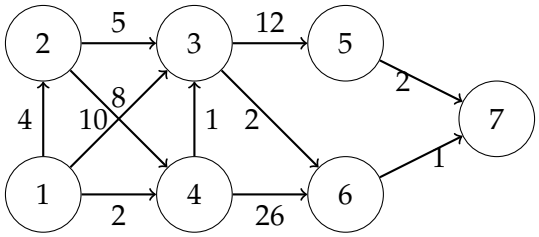

Figure 1: Example of minimum path.

connections between the cities:

Origin	Destination	Cost
1	2	4
1	3	10
1	4	2
2	3	5
2	4	8
3	5	12
3	6	2
4	3	1
4	6	26
5	7	2
6	7	1

We can see the possible connections in Figure 1. First, suppose we start in city 1. Set all values of all cities equal to ∞, except in city 1, which is set to 0. Starting the algorithm, $V = \{\}$, $D = [0, \infty, \infty, \infty, \infty, \infty, \infty]$ (here, this list is ordered by city, where the ith position is the current shortest distance to city i), and C = 1. Obvious, V does not equal all of the cities, so we begin in the loop. We see that the neighbors of 1 are 2, 3, and 4. Currently, D[C] = D[1] = 0, and D[1,2] = 4, D[1,3] = 10, D[1,4] = 2. We also notice that all of these neighbors are NOT in V, so they can be considered. We update the distances of each:

$$D[2] = min(D[1] + D[1,2], \infty) = min(0 + 4, \infty) = 4$$
$$D[3] = min(D[1] + D[1,3], \infty) = min(0 + 10, \infty) = 14$$
$$D[4] = min(D[1] + D[1,4], \infty) = min(0 + 2, \infty) = 2$$

Now put 1 in V, so that $V = \{1.\}$. Now set C equal to the smallest distance. Here, this would correspond to city 4. So we set $C = 4$. Going back up to the loop, we still notice that V does not equal the set of all the nodes. Hence, we continue on. Getting all the neighbors of 4 that are not in V, we have 3 and 6. Updating each of these, we have:

$$D[3] = min(D[4] + D[4,3], 14) = min(2 + 1, 14) = 3$$
$$D[6] = min(D[4] + D[4,6], \infty) = min(2 + 26, \infty) = 28$$

We now put 4 in V, so that $V = \{1, 4\}$, and set C equal to the lowest unexplored node, which

would be 3, so C=3. Again, we see that V is not equal to the set of all the cities, so we continue. We get all neighbors of 3 that are unexplored, which are 5 and 6. We hence update each one:

$$D[5] = min(D[3] + D[3,5], \infty) = min(3 + 12, \infty) = 15$$
$$D[6] = min(D[3] + D[3,6], 28) = min(3 + 2, 28) = 5$$

We now place 3 in V so that $V = 1, 4, 3$. As a reminder, the distances on each city are currently $D = [0, 4, 3, 2, 15, 5, \infty]$. The city unexplored with the shortest distance is currently city 2 with a distance of 4, hence, we set C = 2. Moving to 2, we obtain the neighbors that have not been explored yet (that is, those that are not in V).

$$D[3] = min(D[2] + D[2,3], 3) = min(4 + 5, 3) = 3$$
$$D[4] = min(D[2] + D[2,4], 2) = min(4 + 8, 2) = 2$$

None of the distances have changed, so we still have $D = [0, 4, 3, 2, 15, 5, \infty]$, we add city 2 to V so that $V = \{1, 4, 3, 2\}$. This now leaves us with cities 5, 6 and 7. We will select city 6 (since it has the shortest distance of 5) and set C = 6. We now update the distances of the neighbors of 6:

$$D[7] = min(D[6] + D[6,7], \infty) = min(5 + 1, \infty) = 6$$

We now have $D = [0, 4, 3, 2, 15, 5, 6]$. We now add city 6 to V and we are left with cities 5 and 7. Since 7 is the lowest, we have C=7. Now, city 7 has no neighbors, and so we have no distances to update, and so, we now add city 7 to V so that $V = \{1, 4, 3, 2, 6, 7\}$. This leaves us with city 5, so we set C=5. City 5 has 1 neighbor, so we update the distances:

$$D[7] = min(D[5] + D[5,7], 6) = min(15 + 2, 6) = 6$$

We now add city 7 to V. We notice that V is now equal to the set of all cities, and so, we are done. The final shortest distances to each node are: $D = [0, 4, 3, 2, 15, 5, 6]$. Note that this version of Dijsktra's Algorithm only outputs the shortest path lengths to each city, but note the actual shortest path. A small modification to the algorithm could provide this. We can summarize our output in the following table:

Origin	Destination	Shortest Path Distance	Shortest Path
1	2	4	1-2
1	3	3	1-4-3
1	4	2	1-4
1	5	15	1-4-3-5
1	6	5	1-4-6
1	7	6	1-4-3-6-7

Notice how the Dynamic Programming approach is vastly different than the closed-form approach. We solved this problem by breaking it down into smaller optimization problems (namely, finding the minimum between distances already in the node, and, new distances that are computed). Alternatively, problem could have been characterized as an *binary integer program*. For example, let the decision variable $x_{i,j}$ be equal to 1 if we are going to use that link between the two cities, and let it equal 0 if we are not. To account for the cost of going from city 1 to city

k, we can add together the cost of moving from city i to city j multiplied by the binary variable $x_{i,j}$. If we do not use the connection, then $x_{i,j} = 0$, and the cost $D[i,j]$ will not be added into the total cost (since it would be multiplied by 0). If it is used, it will be added to the total cost (since the cost would be multiplied by 1). The objective function would then be $\sum_{i=1}^{7} \sum_{j=1} 7D[i,j]x_{i,j}$.

We need to ensure that we place constraints on the variables $x_{i,j}$ so that they "make sense". Otherwise, any optimization algorithm we use will return an assignment of all 0's (think about it, the minimum unconstrained of the objective function will just be all 0's to x). We need to ensure the following is true about the relationships between the decision variables:

1. For all cities j, $x_{j,1} = 0$. In other words, we cannot take a path back to the origin city.
2. For all cities j, $x_{k,j} = 0$. That is, once we reach the destination city k, we are done, and should not "leave" this city.
3. For all cities used on the path in between 1 and k, each city can only have 1 incoming city and 1 outgoing city.
4. If a connection does not have a cost assigned to it, we assume the cost is 0. If this is the case, we need to ensure that we only use connections that have a positive cost (the problem can be altered if there are 0 cost paths, but for now, we easily choose this formulation).

Ensuring that these conditions hold true, we should be able to formulate the mathematical program of the same problem. For the first condition, this simply means that we have the constraint $x_{j,1} = 0, \forall j \in \{1,2,3,4,5,6,7\}$. The second condition can be reflected by the constraint $x_{k,j} = 0, \forall j \in \{1,2,3,4,5,6,7\}$. The third condition is a bit more tricky. We can basically break this condition into a more carefully refined statement that will help lead us to the construction of the constraint:

If city i is not city 1 or k, and there exists a m such that $x_{m,i} = 1$, then we must have $\sum_{j=1}^{7} x_{j,i} = 1$. This says that if there is one incoming link into node i, there can only be this incoming link. If the sum adds more than 1, we have two incoming links, and this is not allowed. If it sums to 0, then we reach a contradiction, since we know it is used in at least one city. Likewise, under the same conditions, we need to have $\sum_{j=1}^{7} x_{i,j} = 1$. This says that there must be exactly one city that we move to from city i. If this sum is less than 1, then we don't leave from here and we are done. But this is a contradiction, since it is not the destination city. Likewise, if we have the sum more than 1, then we are leaving to two cities. Again, this is problematic, since we can only move to a single city.

However, if $x_{m,i} = 0$ for all m, then we must have $x_{i,m} = 0$. That is, if we don't move to this city, it is impossible to move to another city from this city. One way to ensure these constraints are true is to introduce a new variable, y_i, which we set equal to 1 if we use the city in the path and set equal to 0 if we do not use the city in the path. Then, we can write the previous constraints as $\sum_{j=1}^{7} x_{i,j} = y_i$. If we do not use city i in the path, then $y_i = 0$, and the constraint says that when we add the x variables, we should get 0. The only way for this to happen is for all the x to be 0, which is what we want. Likewise, if we do use the city, then $y_i = 1$, and the sum of

the x must be 1, which is in line with our previous construction. Likewise, we need the opposite constraint, which we can also modify the same way: $\sum_{j=1}^{7} x_{j,i} = y_i$.

In order to satisfy the last condition, we need to force $x_{i,j} = 0$ when $D[i,j] = 0$. One way to do this is by using the constraint $x_{i,j} \leq D[i,j]$ (if we assume all of the costs are either 0 or some value bigger than or equal to 1, which we do here). Notice that when a cost is not assigned to a connection, we cannot use the connection, and so $D[i,j] = 0$. Since we have this constraint, this would mean that $x_{i,j} = 0$ in this instance. Notice that since we assumed that all costs are either 0 or bigger than or equal to 1, then if $D[i,j] \geq 1$, then the constraint $x_{i,j} \leq D[i,j]$ will be satisfied in either instance of x being set to 0 or 1, since x can only be set equal to 0 or 1. Collecting our observations, we have the following binary integer program:

$$
\begin{aligned}
&\min \sum_{i=1}^{7}\sum_{j=1}^{7} D[i,j]x_{i,j} \\
&\text{s.t.} \\
&x_{i,1} = 0 \ \forall i \in \{1,2,3,4,5,6,7\} \\
&x_{7,i} = 0 \ \forall i \in \{1,2,3,4,5,6,7\} \\
&\sum_{j=1}^{7} x_{i,j} = y_i \ \forall i \in \{1,2,3,4,5,6,7\} \\
&\sum_{j=1}^{7} x_{j,i} = y_i \ \forall i \in \{1,2,3,4,5,6,7\} \\
&x_{i,j} \leq D[i,j] \\
&x_{i,j} \in \{0,1\}, y_i \in \{0,1\}
\end{aligned}
$$

We can solve this model using a *solver*, which will use an optimization algorithm to find the optimal (or sub-optimal, depending on the algorithm) solution. We will revisit this model in the next lecture when we discuss various types of optimization algorithms.

2.6 Discrete vs. Continuous Optimization

There are two main areas of optimization, namely continuous and discrete. Recall from our set theory lecture that sets can be either countable (discrete) or uncountable (continuous). The approach to modeling the real world environment depends on the modeller's choice of how to define the decision variables in their model. If they define a decision to only be assigned to one of a handful of possible alternatives (for example, the specific path to take, or the type of advertisement to use, or the count of employees to promote, etc), then it is a discrete variable. When discrete variables are entered into the mix, this may simplify and be in line with intuition, but paradoxically, the problem becomes more difficult to solve mathematically.

When the modeller uses discrete variables in their program or problem, we can no longer take the same type of approaches to solve the problem. We saw this in the illustration of dynamic programming with the shortest path problem. Notice that no where did we take a derivative, and instead, took an alternative approach to finding the solution. This is common in many types of models that involve discrete variables. We will review through some of the approaches that

are taken to solve these types of problems in the next lecture. However, it should be recognized that discrete optimization is often very different than continuous.

With continuous optimization, the ability to solve these problems is actually "easier", since most of the time we can resort to relying on finding gradients and Hessians. However, even this becomes "difficult" as the size and overall complexity of our models grow. As we had seen in an earlier section, some equations cannot be explicitly solved, and we need to resort to finding *optimality bounds* and other conditions that must hold true in order to (1) show the existence of an optimal solution and (2) characterize the optimal solution. Sometimes finding the gradient may be easy, but the Hessian is another story. Recall that in order to find the second order conditions, we needed to find eigenvalues. This alone can be troublesome if the size of our Hessian matrix is huge (if we are optimizing on say 1000 different continuous variables, this would be a matrix with 1,000,000 elements, and finding the inverse of this may become too computationally burdensome).

The point to take away here is that discrete optimization problems and continuous optimization problems are treated differently. We need to take into account modeling something in the real world as discrete or continuous. While it may "make sense" to treat it as discrete, we possibly loose the ability to even find a solution, or, a "best" solution. If we model something continuously that is in actuality discrete in the real world, then we obtain an increase in our error between our theoretical world and real world. Hence, the modeller needs to be able to identify what can and cannot be modeled as discrete, and if doing so will lead to difficulties in solving the optimization problem.

2.7 Solving Optimization Problems

The first step in optimizing something in the real world is designing the model. This is the challenging part for the business analyst, since they need to gather all of the business requirements, as well as the potential constraints on the decisions that need to be made. This requires a full understanding of everything from the firm's environment, their strategy, their current operations, and the various tactical and operational environments. However, once the modeler iterates through various versions of their model and lands on one that has taken into account all the aforementioned tradeoffs, the next step is to *solve the model*. In otherwords, use the model to find the "best decision".

We have already seen some ways of doing this. If the model is unconstrained and continuous, then we know our first step should be to see if we can compute a gradient, solve for 0, compute the Hessian at that point, and determine the optimality conditions. However, not all problems are treated equally. This approach really only works when the optimization problem is unconstrained, continuous, and first-order partial derivatives exist. But what if any of these conditions fail? Then, we need to take an alternative approach.

Generally, solving any optimization problem is no easy task. However, there are two fundamental ways of finding the optimal solution:

1. **"Work Backward"**: Finding conditions that must be true about the optimal solution, and then solving for it. We have seen this approach so far, and will explore it again in our discussion of constrained continuous optimization.

2. **"Work Forward"**. This can be done in a variety of ways. For example, "Guess and Check", or "Brute Force". If we have discrete elements in our model, we can plug in one by one each

possible solution, compute the objective function (or, if there are also continuous variables, use the first approach to solve the model when the discrete variable is set equal to a value), and choose the discrete value that gives the "best" solution. Needless to say, this approach only really works when the model is small, and the number of possible choices for each decision variable is small.

Some common types of "Work Backward" approaches:

- Finding gradients and Hessians.
- Finding lower and upper bounds.
- Finding equations that must be true about the optimal solution.
- Plotting (bad approach, however!)
- Lagrange Multipliers
- Algorithmic Approaches based on conditions
  - Gradient Descent
  - Newtons Method
  - Penalty Algorithms
  - Leveraging Convexity Properties

Some common types of "Work Forward" approaches:

- Finding Heuristics
- Running a Meta-Heuristic
- Neighborhood Search
- Constraint Relaxation and Tightening
- Estimation of Distribution (EDA)
- Greedy Approaches
- Simulation Approaches
- Brute Force
- Stochastic Algorithms

So which do you use? Well, that decision is again at the discretion of the analyst. It depends on many factors, most of which are actually practical in nature, such as time and budget. The higher budget a firm has to spend on the design of "good algorithms", the better algorithms they will be able to run. For example, brute force approaches can work just fine if you have extreme super computers that just plug in values to the variables, obtain the value of the objective, and store the value, then, take the maximum or minimum. When budget is a concern as well as time, but the firm is willing to accept more error in the approach, then they may resort to a heuristic or meta-heuristic approach, which may not find the "best" solution, but a "good enough" one to make it work. In the next lecture, we will explore some fundamental types of optimization algorithms.

3 Constrained Non-Linear Optimization

In this section, we will review through a set of important results when our optimization model comprises of non-linear equations that also have *constraints*. So far, we have only taken the approach of solving for optimal solutions by finding the gradient and setting it equal to 0 and solving, and then, subsequently finding the Hessian. However, if we have constraints added into the model, then solving for the optimal solution works very differently, since we now need to include the constraints into the model and the approach to solve.

3.1 Fundamentals of Lagrange Multipliers

One way to solve such problems is by the way of *Lagrange Multipliers*. The basic and fundamental idea behind this approach is to take the equations for the constraints and "move them" to the objective function. In other words, we basically convert the constrained optimization problem to an unconstrained optimization problem. There are two ways that this can be done, and it is dependent on if we have equality constraints or inequality constraints, or both. We will begin our discussion with equality constraints, namely by looking at only a single equality constraint. Suppose we have the following mathematical program:

$$\begin{aligned} &\min/\max f(x_1, \ldots, x_n) \\ &\text{s.t.} \\ &g(x_1, \ldots, x_n) = 0 \end{aligned}$$

From this, we can "move" the constraint into the objective function by forming a new function, which is called the *Lagrange Function*, by introducing a variable, λ, called a *Lagrange Multiplier*:

$$L(x_1, \ldots, x_n, \lambda) = f(x_1, \ldots, x_n) - \lambda g(x_1, \ldots, x_n)$$

We can then find the optimal solution by finding the gradient of the Lagrange Function:

$$\nabla_{x_1,\ldots,x_n,\lambda} L(x_1, \ldots, x_n, \lambda) = 0$$

These are the first order conditions for the Lagrange multiplier function. We also have second order conditions as well, however, these are much more intricate than our usual second order conditions. If f and the constraint equation g_i are twice continuously differentiable, and if x^* is a point that satisfies $\nabla_x L(x^*, \lambda^*) = 0$ and $\nabla_\lambda L(x^*, \lambda^*) = 0$ (where ∇_k means to take the partial derivatives with respect to the variables in the vector k), and

$\boldsymbol{y}'\nabla^2_{xx} L(x^*, \lambda^*)\boldsymbol{y} > 0$ for all vectors $\boldsymbol{y}$ such that $\nabla g(x^*)'\boldsymbol{y} = 0$

then x^* is a strict local minimum of f. This is a less restrictive condition, since it only needs to be true on a subset of vectors (namely those that are orthogonal to the gradient of the constraint). If typical second-order conditions still hold in optimization of constrained functions, however, then this condition is by default true. In otherwords, a stronger condition is if the Hessian, taken with respect to only the values of x and not the λ, is positive definite, then it is a minimum, if it is

negative definite, then it is a maximum, if it is neither, then we cannot ascertain any information about it being maximum or minimum.

For example, suppose we want to optimize the function $f(x_1, x_2) = 25 - 2x_1^2 - 3x_2^2$ with the constraint of $-\frac{1}{2}x_1 - x_2 = -5$. We would form the Lagrange function by the following:

$$L(x_1, x_2, \lambda) = 25 - 2x_1^2 - 3x_2^2 - \lambda(-\frac{1}{2}x_1 - x_2 + 5)$$

Now we can find the gradient by finding the partials:

$$\begin{aligned} \frac{\partial L}{\partial x_1} &= -4x_1 + \frac{1}{2}\lambda \\ \frac{\partial L}{\partial x_2} &= -6x_2 + \lambda \\ \frac{\partial L}{\partial \lambda} &= -\frac{1}{2}x_1 - x_2 + 5 \end{aligned}$$

Setting these to 0, we can now form the following system of equations:

$$\begin{aligned} -4x_1 \qquad\qquad &+\frac{1}{2}\lambda = 0 \\ -6x_2 \qquad\quad &+\lambda = 0 \\ -\frac{1}{2}x_1 - x_2 \qquad\quad &= -5 \end{aligned}$$

We can see that the first equation in the system reduces to $x_1 = \frac{1}{8}\lambda$. The second equation reduces to $x_2 = \frac{1}{6}\lambda$. Plugging these into the last equation, we obtain $-\frac{1}{2} \cdot \frac{1}{8}\lambda - \frac{1}{6}\lambda = -5$. Solving for λ, we obtain $\lambda = \frac{240}{11}$. Plugging this in, we find that $x_1 = \frac{30}{11}$ and $x_2 = \frac{40}{11}$.

Now we will find the Hessian to determine if the sufficiency conditions hold to determine if it is a minimum or maximum (recall that this method will NOT always work in the case of constrained optimization, and we may need to resort to showing that it is positive/negative definite on only a subset of vectors that are orthogonal to the constraint). Taking the second order partials, we have:

$$\begin{aligned} \frac{\partial^2 f}{\partial x_1 \partial x_1} &= \frac{\partial}{\partial x_1}[-4x_1] = -4 \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} = \frac{\partial^2 f}{\partial x_1 \partial x_2} &= \frac{\partial}{\partial x_1}[-6x_2] = 0 \\ \frac{\partial^2 f}{\partial x_2 \partial x_2} &= \frac{\partial}{\partial x_2}[-6x_2] = -6 \end{aligned}$$

Forming the Hessian at $(x_1, x_2) = (\frac{30}{11}, \frac{40}{11})$, we have:

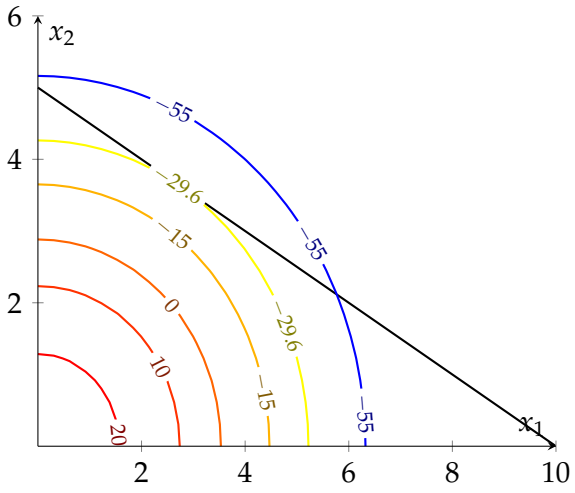

Figure 2

.

$$H(\frac{30}{11}, \frac{40}{11}) = \begin{bmatrix} -4 & 0 \\ 0 & -6 \end{bmatrix}$$

Finding the eigenvalues gives us:

```
>>> A=np.array([[-4,0],[0,-6]])
>>> A
array([[-4,  0],
       [ 0, -6]])
>>> ev,evec=np.linalg.eig(A)
>>> ev
array([-4., -6.])
```

Since both of them are negative, we can see that the Hessian is negative definite, which implies the second order sufficiency conditions for constrained optimization. Hence, we have a maximum. We can see this visually in the plot. The value of the maximum is $f(\frac{30}{11}, \frac{40}{11}) = 25 - 2(\frac{30}{11})^2 - 3(\frac{40}{11})^2 = \frac{3025}{121} - \frac{1800}{121} - \frac{4800}{121} = \frac{-3575}{121} \approx -29.6$. We can see from Figure 2 how this is the maximum. The black line is the line $-\frac{1}{2}x_1 - x_2 + 5 = 0$, which is the collection of (x_1, x_2) points that are under consideration. That is, the only points for the function $f(x_1, x_2)$ that are feasible to consider are those on the black line in the figure. We can see the contour plot of the function $f(x_1, x_2)$ at different levels. When $f(x_1, x_2) > -29.6$, we see that the curves do not intersect the black line at all, and so these (x_1, x_2) values are not feasible to consider. We notice that the feasible solutions are those on the black line, and the (x_1, x_2) values for the function such that $f(x_1, x_2) \leq -29.6$ do intersect the black line twice. However, notice that the yellow line is the greatest of these values, which is on the black line at $(\frac{30}{11}, \frac{40}{11})$. When we consider any other (x_1, x_2) point on the black line, the value of $f(x_1, x_2)$ (represented by the colored lines) is strictly less than -29.6. This illustrates our mathematics, and shows that the maximum does indeed occur at $(\frac{30}{11}, \frac{40}{11})$.

3.2 The Karush-Kuhn-Tucker (KKT) Optimality Conditions

We can generalize the idea of Lagrange Multipliers by considering not only a single equality constraint, but also multiple equality and multiple inequality constraints. When this is the case, we slightly change our optimality conditions that we need to test to determine what the optimal solution is. When we do this, these conditions are referred to as the *Karush-Kuhn-Tucker*, or KKT, conditions. The first order conditions are stated as follows. Consider the problem

$$\begin{aligned}
&\min f(x_1, \ldots, x_n) \\
&\text{s.t.} \\
&g_1(x_1, \ldots, x_n) = 0 \\
&g_2(x_1, \ldots, x_n) = 0 \\
&\vdots \\
&h_k(x_1, \ldots, x_n) \leq 0 \\
&h_1(x_1, \ldots, x_n) \leq 0 \\
&h_2(x_1, \ldots, x_n) \leq 0 \\
&\vdots \\
&h_m(x_1, \ldots, x_n) \leq 0
\end{aligned}$$

If x^* is a local minimum of this them, and if f, g_i, h_j are all twice continuously differentiable, and if certain regularity conditions hold true, then there exist unique multipliers $(\lambda_1, \ldots, \lambda_k)$ and $(\mu_1, \ldots, \mu_m)$ such that the gradient of the Lagrangian function

$$L(x_1, \ldots, x_n, \lambda_1, \ldots, \lambda_k, \mu_1, \ldots, \mu_m) = f(x_1, \ldots, x_n) + \sum_{i=1}^{k} \lambda_i g_i(x_1, \ldots, x_n) + \sum_{j=1}^{m} \mu_j h_j(x_1, \ldots, x_n)$$

yields a minimum value of the function. In other words,

$\nabla_x L(x_1^*, \ldots, x_n^*, \lambda_1^*, \ldots, \lambda_k^*, \mu_1^*, \ldots, \mu_m^*) = 0$ and $\mu_j^* \geq 0$ $\mu_j^* = 0$ for any j such that $h_j(x_1, \ldots, x_n) \neq 0$

$y' \nabla_{xx} L(x_1^*, \ldots, x_n^*, \lambda_1^*, \ldots, \lambda_k^*, \mu_1^*, \ldots, \mu_m^*) y \geq 0$ for all vectors y such that $\nabla g_i(x_1^*, \ldots, x_n^*)' y = 0$ and $\nabla h_j(x_1^*, \ldots, x_n^*)' y = 0$ for all j such that $h_j(x_1^*, \ldots, x_n^*) = 0$.

In otherwords, we can take a similar approach of finding partial derivatives of the Lagrangian function, solving for all the multipliers, and then finding a critical point. The Hessian can then be used to determine if the critical point is a minimum or a maximum.

4 Linear Programming

When a mathematical program includes only linear constraints and a linear objective functions, then the program can be solved in a different manner as we discussed so far. The resulting program, by the way, is referred to as a *Linear Program*. We also assume that all of the variables

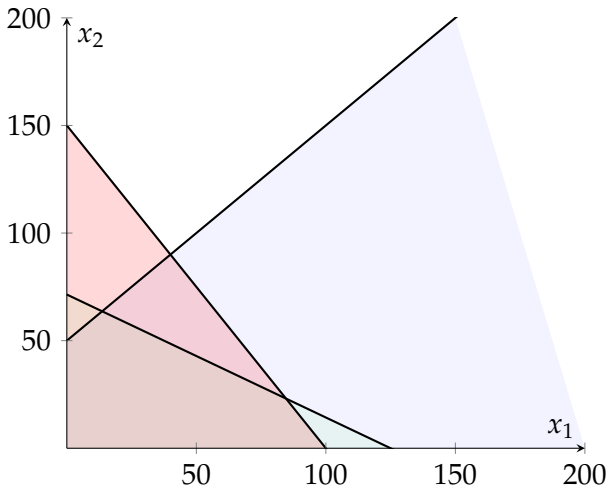

Figure 3

.

are continuous variables. However, we will see in the next lecture that the method of solving these models rests on a well-known algorithm referred to as the *Simplex Algorithm*. In this section, we will illustrate more of the geometry regarding Linear Programs. Consider the following linear program:

$$\begin{aligned} &\max\ 5x_1 + 5x_2 \\ &\text{s.t.} \\ &3x_1 + 2x_2 \leq 300 \\ &-x_1 + x_2 \leq 50 \\ &4x_1 + 7x_2 \leq 500 \end{aligned}$$

We can see from Figure 3 the constraints visualized. The shaded regions are the values of (x_1, x_2) such that the constraint is true. When we are optimizing a linear program, we need *all* the constraints, not just one or two, to be true. This region is referred to as the *feasible region*, and geometrically, it is the intersection of all the areas shown in Figure 3. This intersection is shown in the blue region in Figure 4, along with the contour lines of the objective function. We can make a few observations. First, the optimal solution is near $(x_1^*, x_2^*) = (85, 23)$, at a value of about 539.

We can see that if we take any more contour lines outward, we leave the feasible region, and hence, those solutions do not matter. We also notice that the optimal solution occurs on a point of intersection between two of the constraint lines in the feasible region. These points are called simplexs. Generally speaking (which we will state without proof), if we have a linear program, and an optimal solution exists, then the solution is one of the simplexs. This leads us to the *simplex algorithm*, which essentially "moves" from one simplex to the next, testing the values of the objective function, and eventually stopping at the "best one".

Obviously, if we are solving Linear Programs with higher dimensions, it becomes more chal-

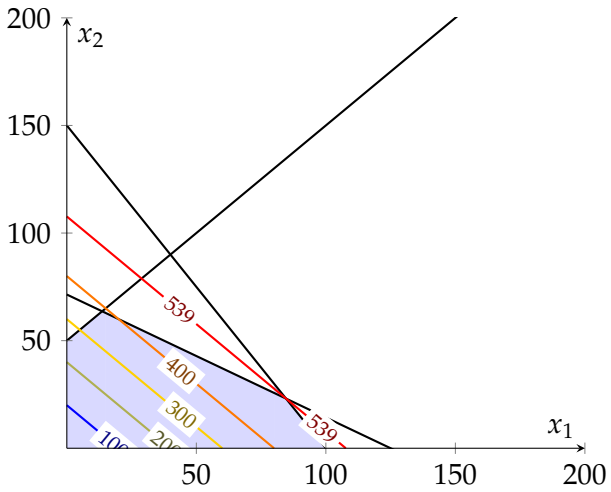

Figure 4

.

lenging to visualize these. However, given our observations, an algorithm which "moves" from simplex to simplex, even in higher dimensions, will help us solve the problem somewhat quickly. We also illustrated earlier how linear programs can be formulated. Formulation is the most important part of the modeling process. However, the next important is the solution methodology, and this depends on the complexity of the problem itself, a topic which we will discuss in the next section.

5 Measuring Problem Difficulty

5.1 Why We Care About Problem Difficulty

So far, we have explored how the analyst can formulate a model of the real world, and have even illustrated some *algorithms* in order to solve these models, or problems. However, an important consideration for the analyst is just how "hard" their problem is to solve, if it can even be solved. If their problem is "harder" than other known "hard" problems to solve, then the analyst is at a crossroads, since there may be no known way to solve their existing problem. In these instances, the analyst needs to most likely rework their model, or change their question as to what they would like to solve. This almost always amounts to them having to sacrifice any constraints on error they may have had in exchange for solvability.

There is an old expression in this regard, namely, that the analyst starts where the computer scientist stops. Computer scientists are often concerned about problems that they can solve. If they have found that it can't be solved, they typically move on. However, the analyst is in a different situation, since their problem has extreme practical relevance. As such, what commonly occurs when the analyst comes across a "hard" problem is that they change their question or model around. For example, they may ask a question such as "what is the highest profit we can possible earn with all of our constraints?". If they find that the answer to this question is too difficult to find, they may resort to a different question, like "does there exist an approximate profit that will be higher than many other reasonable decisions explored?" or "does there exist a "good enough" solution that we can live with?", or "does there exist an optimal solution that we

can find in a reasonable amount of time if we remove this constraint?".

The point here is that yet another tool in the analyst's box is the ability to effectively understand how "hard" their model is to solve. Knowledge of this helps guide the analyst to determine the best *algorithm* to use to solve their model, or, the specific changes they need to make so they can apply an existing algorithm. The level of difficulty that a problem is to solve is referred to as it's *complexity*. There are many different *classes* of complexity which categorize problems as being "hard" or "easy", or somewhere in-between. In order to have an understanding of this concept, we first need to delve into some fundamental concepts.

5.2 Definition of Problems and their Properties

The ability to measure how "hard" a problem is rests on a lot of technical mathematical theory (primarily on set theory, abstract algebra, and formal languages and automata), all of which we will avoid in this discussion. Instead, we will provide a brief overview into more of the practical considerations in measuring the "difficulty" of a problem. To begin, we first need to define a "problem". Put simply, a problem is a question. A solution to the problem is a "correct" answer to the question. There are two fundamental types of problems: *decision problems*, which can be solved by providing a "yes" or "no" answer, and *non-decision* problems, where the answers are much more complex than a simple "yes" or "no". If we can, we try our best to always reformulate a problem to a decision problem, although this is not always possible.

For example, we may ask, "what is the best decision for a firm to make so that they maximize their profit?". This is a *non-decision* problem, since the possible output of the question is a number or collection of numbers. We may instead ask "If we order $Q = \sqrt{\frac{2K\lambda}{h}}$, is this the optimal solution?". Notice that the only possible answers to this question are "yes" or "no". Most problems we solve in practice are actually not inherently decision problems. However, the way in which we characterize "difficulty" is in terms of decision problems. When given any problem P , regardless of it's type, we are often concerned with two types of questions that arise out of the problem. The first is, "can we find a solution to the problem?" (notice that this question is itself a decision problem). This is called the *solveability* of the problem, and it speaks to the ability to construct an *algorithm* which can run at a certain level of performance to *solve* the problem. The second type of question is "if we are given an answer to the problem, can we *verify* that the solution is correct?". This is called the *verifiability* of the problem, and it again speaks to the ability to "guess and check" possible solutions to the problem in a "reasonable" amount of time.

Hence, for any problem or model we encounter, we always ask "can it be solved in a reasonable amount of time?", and if not, "can it be verified in a reasonable amount of time?". Notice that both of these questions, regardless of the type of underlying problem, are decision problems. For example, consider the popular *traveling salesman problem*. The problem states that given n cities, where there is a distance between each city, can we find the shortest distance *tour* (which is a path that starts at the starting city and ends at the starting city) that visits each city only once? This is **not** a decision problem as stated, but can be reworked into one. For example, we can ask, "given a total distance D, does there exist a tour that visits each city which it's total distance is less than D?". This is now a decision problem, since the answers are only "yes" or "no".

This problem has no known algorithm which can be ran within a *reasonable* amount of time to answer the question. However, if we are given an *instance* of the problem and a solution, then we can certain verify if the solution is correct or not in a *reasonable* amount of time. If we have n

cities and we are given a specific tour and a specific value for D, then we can add together the distances between each city in the given solution and compare it against D. This can be done in a "reasonable" amount of time, since we only need to perform $(n-1)$ computations (more specifically, addition of the distances between each city in the given solution) and one additional computation for comparison against D, which makes the verification process conducted in n steps. Since we cannot find, for the time being at least, a *solution* to the decision problem, it is not solvable (or more appropriately called decidable), but it is verifiable.

5.3 Algorithms and Computational Complexity

In order to discuss the "difficulty" of a problem, we often refer to the ability of the "best" known algorithm which can either decide or verify the problem. Hence, we need to know something about how we measure the performance of an algorithm, and, more importantly, the definition of an algorithm. Put simply, an algorithm is a list of steps that takes inputs and converts them to outputs in a certain way. Algorithms can be *deterministic* if it provides the same output everytime the algorithm runs for the same inputs. On the otherhand, if the algorithm takes different "paths" to run each time, it is possible that the outputs may differ on different runs of the algorithm with the same inputs.

There are many ways to measure how "good" an algorithm can run. We commonly measure the quality of an algorithm by first understanding the number of computations the algorithm performs as a function of the *input size*, which indicates the number of the inputs provided to the algorithm. For example, an algorithm to square a number has an input size of 1, which multiplying two numbers has an input size of 2. An algorithm to add n numbers has an input size of n, while an input size to find the determinant of an $n \times m$ matrix is nm.

Studying an algorithm involves "counting" the number of steps it performs, typically as a function of the input size. For example, we can have two algorithms which compute the sum from 1 to a given input n. The first algorithm may be to perform $(n-1)$ additions iteratively. Hence, the algorithm will perform in a total of $n-1$ steps. A second algorithm would be to use Gauss' equation to compute this, which we know by now is $\frac{n(n+1)}{2}$. This algorithm is performed in only 2 steps (first multiply the two numbers in the numerator, and then divide by two). We are solving the problem in two different ways, using two different algorithms, both of which perform differently.

However, counting the steps is not always obvious, since a high-performing algorithm may terminate with a solution much earlier than our "count". For example, suppose we want to construct an algorithm to determine if there are at least 5 people who own a dog in a collection of data. The answer to this question is clearly "yes" or "no". If we provide an input of n data points, then we have two situations in which the algorithm will perform. The first is that the first 5 people all have dogs, and so, in the *best case*, the algorithm will terminate after 5 iterations. Clearly, this is less than n (if we assume the data is always greater than 5 people). On the other hand, our data may be sorted in a way such that if we have exactly 5 people who own dogs, the last person to own a dog is also the last person on the list. Hence, we would need to run the algorithm, at the *worst case*, a total of n times to make this determination.

We have hence illustrated that characterizing algorithms based on the exact number of computations it will perform may not be an appropriate approach, and so we instead want to focus on characterizing algorithm performance by studying the *best-case performance* and *worst-case performance*. Overwhelmingly, we study the worse-case performance. Hence, given an algorithm

with an input size of n (and, possibly other input sizes as well), the idea is that we would like to find a function $f(n)$ of the input such that it will characterize the number of computations. Furthermore, we notice that the value of $f(n)$ will change as we increase the input size. Moreover, this value will most likely become larger as n becomes larger (most algorithms will always require additional computations for larger inputs).

The way in which we characterize "worst-case performance" is by studying the behavior of the function $f(n)$ as $n \to \infty$. In otherwords, we want to characterize how "long" the algorithm will take if we continue to increase the input size to the algorithm. One way to make this characterization is through the use of the *Big O Notation*. The idea is simple. If we can find a function $f(n)$ that exactly characterizes the number of steps the algorithm will perform, then we want to find a different function $g(n)$, which is typically easier and more generic to specify, which will characterize an upper bound to the function $f(n)$ for all n bigger than a given N. More specifically, we say that $f(n) = O(g(n))$ if there exists a constant c and an integer N such that $f(n) \leq cg(n)$ for all $n > N$.

For example, if we found that our algorithm will run at worst case in $f(n) = n^2 + 2n + 3$ steps, then we say that our algorithm runs in $O(n^2)$ time, since it is possible to find a constant c and an integer N such that $n^2 + 2n + 3 \leq cn^2$. Generally, whenever we have any polynomial function $f(n)$ of degree m, that is, if $f(n) = a_0 + a_1x + a_2x^2 + \cdots + a_mx^m$, then $g(n) = x^m$ (prove this!). For example, suppose we have the algorithm:

```
procedure add_square_matrix_entries (matrix_size,matrix_A)
set total_sum = 0

for each i in 1 to matrix_size
    for each j in 1 to matrix_size
        total_sum = total_sum + A[i,j]

return total_sum
```

Let us count the number of steps. First, we have the assignment of total_sum to 0. This counts as one computation. We then have for each i, j permutation 2 computations. The first computation is the addition of total_sum and the entry in the matrix at row i and column j. The second computation involves taking this number and assigning it back to the variable total_sum. Therefore, for each unique permutation (i, j), there are 2 computations. If we let n be the number of rows and columns, then the total number of permutations that the algorithm will explore in the double loop is n^2 (for a single value of i, we iterate over n values of j, and since there are n values for i, this gives us a total of $nn = n^2$ iterations). Since each iteration has 2 computations, this means that this part of the algorithm will perform $2n^2$ computations. Hence, the total number of computations in the algorithm would be the 1 computation for assignment and the $2n^2$ computations for the double loop. This means that $f(n) = 2n^2 + 1$. In big O notation, this means that our algorithm will perform with a complexity of $O(n^2)$, at it's worst case.

5.4 Complexity Classes

Now that we have an idea on how to measure the performance of an algorithm, we can classify how "difficult" certain problems are based on the "best known" performing algorithm to solve the problem. We say "best known", because much of these classifications are actually not known with certainty. We will discuss briefly an open problem that still exists to this day in regards

to computational complexity. But it is important to keep in mind that when we say a problem belongs to a certain set of "difficulty", this is purely defined based on what we know right now. Things may change in the future with the discover of better performing algorithms, and hopefully, one day, we can solve the unknown problem. So keep in mind, if we say that a problem is "verifiable but not decidable", we simply mean this within the context of what is known *right now*.

In other words, we may have a decision problem that we can verify, but cannot decide (like the decision version our Traveling Salesman Problem). This does not mean that the problem is unsolvable in an unreasonable amount of time. It just simply means that it is unsolvable with the *currently known* algorithms in existence. It is entirely possible that an algorithm for a problem which was originally undecidable (like the decision version of Traveling Salesman Problem) can be deemed decidable at a later point in time. Although, if you prove this, or find an algorithm that can decide this, you would pretty much never need to do math again, since the prize money for one of these unknown problems is currently at $1,000,000!

Getting back around to our discussion on problem classification, we would like to construct a classification system, defined in terms of mathematical sets, that can classify the problems in certain "levels of difficulty". Typically, we do this based on two important criteria. First, we typically classify *decision problems*. If our problem is not a decision problem, we try to find a way to convert, or reduce, or transform, our original problem into a simple "yes" or "no" answer. This is not to say that a problem which is not a decision problem cannot be classified, but, to simplify our discussion, we consider only those problems that can be transformed into a simple "yes" or "no" output. Second, "good performing" algorithms are algorithms that run in *polynomial time*. In other words, if algorithms can run in $O(n^k)$ for some k.

Hence, the way we will construct our classification system is based on "good performing" and "not good performing". However, it becomes more intricate than this. Recall that we said that any problem can be asked two important questions: can it be verified, and can it be decided? If the answer is an affirmative "yes" for both of these, then we have a class of problems which can be solved and verified in "reasonable time" (heretofore, known as polynomial time). We hence have our first class of problems that we would like to construct. We say that the class $\boldsymbol{P}$ is the collection of all problems where there exists an algorithm that can run in polynomial time which can decide (or answer) the question with a "yes" or "no" and there exists an algorithm that can run in polynomial time which can verify the question for a given solution. The first condition actually implies the second (if we can solve it, then we can verify it), but the converse is not necessarily true (it is not true in general that if we can verify it, we can solve it).

By definition, all problems in $\boldsymbol{P}$ are deterministic. That is, the algorithms that provide the solution for a given input will produce the same output (yes or no) for the same input every time. But what about problems that we can verify, but not decide (i.e. the decision problem version of TSP)? Well, we now have yet another set of problems to consider. We say that the class $\boldsymbol{NP}$ is the collection of all problems that can be verified in polynomial time, but we do not know if there exists an algorithm that can decide the problem in polynomial time. **Do not confuse this class**. Many people often attribute the $\boldsymbol{N}$ in the $\boldsymbol{NP}$ to mean "non-polynomial". **This is not true.**. The abbreviation actually stands for "non-deterministic polynomial".

Based on our definitions, we should clearly see that $\boldsymbol{P} \subseteq \boldsymbol{NP}$. Be careful here. We put the equal sign underneath for a reason: this is still an solved problem (more on this later). It should be easy to see why the collection of all "easy problems" are a subset of "not-so-easy problems".

If $\boldsymbol{NP}$ is defined as the set of problems which can be verified in polynomial time, then we know that the problems in $\boldsymbol{P}$ need to also be in $\boldsymbol{NP}$, since a problem which can be decided in polynomial time can also be verified in polynomial time. Hence, all elements of $\boldsymbol{P}$ are in $\boldsymbol{NP}$, which means $\boldsymbol{P \subseteq NP}$

The way that we think about "problem difficulty" is by thinking about it from a relative standpoint. If we can show that a problem is "at least as hard" as a different problem which we know must belong to a particular class, then the problem at hand must be either in a "more difficult" set, or the very lest, the same set. Such problems are referred to as $NP - Hard$ problems, which are problems that are at least as "hard" as $\boldsymbol{NP}$ problems. $NP - Hard$ problems can be either $\boldsymbol{NP}$ or not $\boldsymbol{NP}$. For example, the traveling salesman problem (the non-decision version of it) is a purely $NP - Hard$ problem. Currently, there does not exist an algorithm that will run in polynomial time to verify that you have the shortest tour while visiting every city once. If given a specific tour, how can we verify that it is the shortest distance tour? How do we know that there is not another tour smaller than the one provided to us? Currently, the only way to verify a given tour is by running algorithms that do **not** run in polynomial time, and hence, by definition, the TSP would **not** be in $\boldsymbol{NP}$.

The decision version of TSP however IS in $\boldsymbol{NP}$ AND in $NP - Hard$ (this sounds strange, but just stick with me for a moment. The TSP is in $NP - Hard$ because we know that it's verification problem is *at least as hard* as all other $\boldsymbol{NP}$ problems (that is, it is at least as hard as the "hardest problem" in $\boldsymbol{NP}$. If a problem is not "at least as hard" as all the other problems in $\boldsymbol{NP}$, but is not in $\boldsymbol{P}$, then it is purely in $\boldsymbol{NP}$. However, for TSP, we know it is at least as hard as the hardest problems in $\boldsymbol{NP}$. When we have this situation, where a problem is both $NP - Hard$ and $\boldsymbol{NP}$, we say that it is in $NP - Complete$. **DO NOT CONFUSE THE TSP PROBLEM HERE!** As it turns out, the raw version of TSP is $NP - Hard$, and NOT in $\boldsymbol{NP}$ (again, how can we verify that a given tour is the "best" tour in polynomial time? Right now, we cant, hence, it's in $NP - Hard$). The decision version of TSP, however, is both $NP - Hard$ AND $\boldsymbol{NP}$, which means, it's $NP - Complete$. Put simply, the raw version of TSP is $NP - Hard$ while the decision version of the problem is $NP - Complete$. But these two versions are not "equivalent". That is, just because we can verify the decision version of the problem does **not** mean we can verify the non-decision-version of the problem. Keep this in mind!

5.4.1 A Word on NP and P

While the classes we have discussed we have defined somewhat loosely, they do have rigorous definition which we did not explore, and will not explore. Despite this, even with rigorous definitions, the class that a problem belongs to essentially comes down to whether or not a polynomial algorithm to verify or decide exists. Existence is somewhat challenging to prove in this context. We have mentioned, however, that if a problem belongs to the class $\boldsymbol{P}$, then by definition, it must also belong to $\boldsymbol{NP}$. In otherwords, $\boldsymbol{P \subseteq NP}$. But what about the opposite?

In other words, if we have a problem that is verifiable, can it also be decidable (that is, if we can verify it, can we also decide it?). Believe it or not, this is still an open question and unsolved problem. If it is true that $\boldsymbol{NP \subseteq P}$, this would imply that $\boldsymbol{P = NP}$, which essentially means that all problems which we can verify, we must also be able to find an algorithm which we can decide. This would indeed have very deep implications for optimization problems, and computer science more generally.

The current state of this problem, however, remains that most computer scientists believe

that $\boldsymbol{NP} \neq \boldsymbol{P}$, although a proof for this has still not been found. By the way, a valid proof or counter-example would land you top fame, and a nice $1,000,000. For now, let's focus on some problems that are solveable!

5.4.2 Reducability

At this point, you may be wondering, how can we possibly show that a problem is within one class or another? That is, how can we prove that a problem is in $\boldsymbol{P}$, $\boldsymbol{NP}$, $NP-Hard$, or $NP-Complete$? We could try to prove this directly, but this is often too difficult and cumbersome. Instead, when we are faced with a problem, what we try to do to determine it's complexity is *reduce* another well known problem for which we know the class membership to our problem. How do we reduce a problem?

If we can find an algorithm which runs in polynomial time that will convert another problem into our problem, and we know that the other problem belongs to one class or the other, then we know our problem must also belong to that class. For example, suppose we wanted to solve the following problem: "Consider n cities, where there is a cost to travel in between each city. Assume that each city is connected in such a way that there exists at least one tour (starting at the same city you end at, visiting each node exactly once), and that each city can be visited in both directions. What is the minimum cost path ending at the same city it begins in such that exactly one city in the path, other than the starting and ending city, is visited exactly twice, but every other city is visited exactly once?"

You may immediately think, "oh, this is TSP!". Well, no, it is not. Our problem requires us to visit one of the cities exactly twice on the path, while the other cities can only be visited once. However, if we can convert TSP to our problem, such that the inputs of TSP are mapped to the inputs of our problem for a "yes" instance of our problem and TSP, and the outputs of TSP are mapped to the outputs of our problem (ie "yes"), and if we can show how to perform this mapping in polynomial time, then we have shown that solving our problem is "just as hard" as solving TSP.

In other words, we basically try to reduce our problem to trying to solve a TSP problem by constructing a polynomial algorithm to "convert" our inputs which will yield "yes" instances to TSP "yes" instances. If we can do that, then we have shown our problem is "equivalent" to solving TSP, and hence they would have the same complexity (this result comes from a LOT of theory which I have chosen to not demonstrate here since it is somewhat out of the scope of this discussion). Let us try to do this with our problem and TSP:

Assume that $n_1 - n_2 - \cdots - n_i - \cdots - n_i - \cdots - n_1$ is the minimum distance path in our problem where node n_i is visited twice. If we introduce a new node n_{n+1} such that we re-designate one of the n_i to relabel it as n_{n+1}, and make all the connections to other cities that n_i has, then we obtain a graph with $n+1$ distinct cities. If we make the assignment of the path in our original problem of $x_i = x_i$ for all $i < n+1$ and $x_{n+1} = n_{n+1}$, then we obtain a tour in the newly converted problem. This path contains all cities in the converted problem, each city is distinct, and this path is also the minimum distance tour in this new constructed problem. If it were not the minimum distance tour, then we would obtain a different sequence of the x's, but then this would map to a different path in our original problem that would also be minimum in our problem, since all the weights are the same, which would be a contradiction. Hence, our "yes" instance to the original problem maps to a "yes" instance in the TSP problem. In short, we have reduced the verifiability of our problem to the verifiability of a TSP. Furthermore, these

assignments all occur in polynomial time, and so our algorithm to make this conversion occurs in polynomial time. By transitivity, this means that our problem can be reduced to a TSP problem. Since TSP is NP-Hard, so too must be our problem.

In summary, to prove that a problem is in a certain class, we pick a known problem to essentially convert our problem to. If we can show that a sub-problem, or the entire problem of our problem can be restated as a well-known problem, then we know it is at least that level of complexity. There are many "well-known problems" to pick from. Here are just a few:

1. 3-SAT (the most famous one!)
2. 3-Color
3. 3DM
4. Vertex Covering
5. Exact Covering
6. Planar-3-Color
7. The Clique Problem
8. Hamaltonian Cycles
9. Partitioning Problem
10. Knapsack Problem
11. Bin-Packing Problem
12. Integer Programming
13. Hamiltonian Path
14. Independent Sets
15. Traveling Salesman (TSP)

A good analyst spends a bit of time learning all of these algorithms to understand how they are formulated, how they are commonly solved (if they can be solved), and, to use as a "benchmark" for new problems. In otherwords, part of the analyst's "toolbox" is knowing a handful of well-known problems which belong to certain complexity classes. If you have a grasp on these, then any new problem that comes your way is essentially an "extension" of a well-known problem. For example, if you are trying to find an optimal truck route such that each city is visited at most once and the truck needs to pick up and drop off products, if you need to determine what the route is, as well as how many products to drop off and pick up, then these this type of problem can clearly be converted to a TSP problem (with some adjusting of the costs of pickup and drop off), which means this problem is also NP-Hard.

We will not review all of these problems: this is best left to the analyst to study on their own! The point to take away here: when you have a new problem, you need to figure out it's complexity. Once you know it's complexity, you have an idea of which algorithmic approach to take. If it

can be verified and decided within polynomial time, and you have proven this by showing it is reducible to another problem, then you can run a similar algorithm on that converted problem. If you know the problem is NP-Hard, then you are out of luck, as no algorithm exists to run it in polynomial time. The only way your can solve it is: run an exponentially timed algorithm on a distributed computing system if the problem instance size is small enough to handle, or: design a heuristic or meta-heuristic to "guess and check", in a "smart" way, as we will see in our next lecture.