# Linear Programming

Linear programming is one of the powerful tools that one can employ for solving optimization problems. This technique has proven to be of value in solving a variety of problems that include planning, routing, scheduling, assignment and design.

The purpose of this note is to describe the value of linear program models. It is not to describe in detail the algorithms used to solve linear programs.

## 1 Example: A Simple Manufacturing Problem

Suppose that a company that produces three products wishes to decide the level of production of each so as to maximize profits. Let $x_1$ be the amount of Product 1 produced in a month, $x_2$ that of Product 2, and $x_3$ that of Product 3. Each unit of Product 1 yields a profit of 100, each unit of Product 2 a profit of 600, and each unit of Product 3 a profit of 1400. There are limitations on $x_1, x_2$, and $x_3$ (besides the obvious one, that $x_1, x_2, x_3 \geq 0$). First, $x_1$ cannot be more than 200, and $x_2$ cannot be more than 300, presumably because of supply limitations. Also, the sum of the three must be, because of labor constraints, at most 400. Finally, it turns out that Products 2 and 3 use the same piece of equipment, with Product 3 using three times as much, and hence we have another constraint $x_2 + 3x_3 \leq 600$. What are the best levels of production?

The total profit is represented by the expression $100x_1 + 600x_2 + 1400x_3$ and this is the function that we seek to maximize, and this is also called the *objective* function. The constraints are as described in the problem setup.

The *linear program* is then represented as

$$\text{maximize } 100x_1 + 600x_2 + 1400x_3$$

subject to the constraints

$$\begin{aligned} x_1 &\leq 200 \\ x_2 &\leq 300 \\ x_1 + x_2 + x_3 &\leq 400 \\ x_2 + 3x_3 &\leq 600 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

A vector $\{x_1, x_2, x_3\}$ that satisfies the constraints is called a *feasible* solution. The *optimal solution* is one of the feasible solutions. The set of all feasible solutions forms a polyhedron in three-dimensional space.

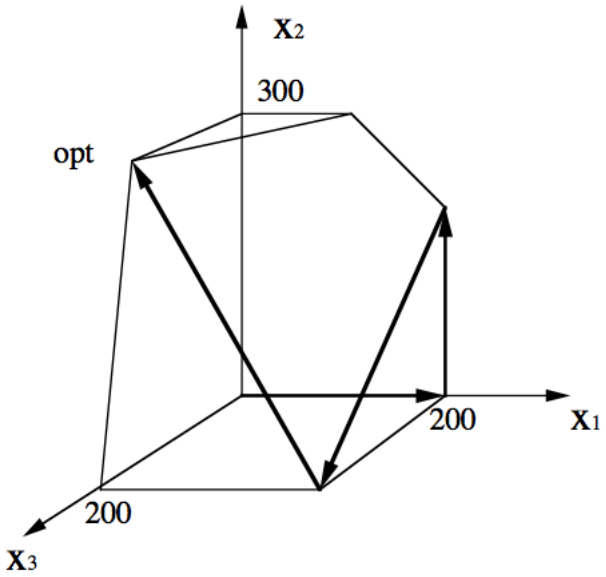


Figure 1: A geometric visualization of the constraints and objective function for the linear program in Section 1.

We wish to maximize the linear function $100x_1 + 600x_2 + 1400x_3$ over all points of this polyhedron (Figure 1). Geometrically, the linear equation $100x_1 + 600x_2 + 1400x_3 = c$ can be represented by a plane parallel to the one determined by the equation $100x_1 + 600x_2 + 1400x_3 = 0$. This means that we want to find the plane of this type that touches the polyhedron and is as far towards the positive orthant as possible. Obviously, the optimum solution will be a vertex (or the optimum solution will not be unique, but a vertex will do). Of course, two other possibilities with linear programming are that (a) the optimum solution may be infinity, or (b) that there may be no feasible solution at all. Any standard method for solving an LP will detect unboundedness or the non-existence of solutions.

## 2 Linear Programs

Linear programs, in general, have the following form: there is an objective function that one seeks to optimize, along with constraints on the variables. *The objective function and the constraints are all linear in the variables; that is, all equations have no powers of the variables, nor are the variables multiplied together.* As we shall see, many problems can be represented by linear programs, and for many problems it is an extremely convenient representation.

If one can solve linear programs (LPs) in general then the central question is how an optimization problem can be reduced to an LP formulation.

There are polynomial-time algorithms for solving LPs. One measures the size of an LP using the number of variables ($n$) and the number of constraints ($m$) thus a polynomial-time algorithm has a running time that is a polynomial in $n$ and $m$.

Polynomial-time solutions to LPs rely on *interior-point methods* that were initially suggested by Khachiyan and made efficient by Karmarkar. More often, however, LPs are solved using the *Simplex method* devised by Dantzig. The Simplex method is not a polynomial-time algorithm (it can take an exponential amount of time in some cases) but it solves many problems quickly and recent work on the smoothed analysis of algorithms by Spielman and Teng explains this behaviour.

The simplex method starts from a vertex of the polytope that represents the feasible region and repeatedly looks for a vertex that is adjacent, and has better objective value. That is, it is a kind of hill-climbing in the vertices of the polytope. When a vertex is found that has no better neighbour, simplex stops and declares this vertex to be the optimum. (This is illustrated in Figure 1. The directed lines, starting at (0,0,0), indicate some choices that the Simplex method may make.) There are now implementations of simplex that routinely solve linear programs with many thousands of variables and constraints.

## 3 Example: Production Scheduling

We have the demand estimates for our product for all months of 2011, $d_i, i = 1, 2, \ldots, 12$, and they are very uneven, ranging from 4400 to 9200. We currently have 30 employees, each of which produce 200 units of the product each month at a salary of 5000; we have no stock of the product. How can we handle such fluctuations in demand? Three ways:

- Overtime: but this is expensive since it costs 80% more than regular production, and has limitations, as workers can only work 30% overtime.
- Hire and fire workers: but hiring costs 2000, and firing costs 4000.
- Store the surplus production: but this costs 80 per item per month.

This problem can be formulated and solved as a linear program. As in all such reductions, the crucial first step is defining the variables:

- Let $w_i$ be the number of workers we have the $i^{th}$ month; we have $w_0 = 30$.
- Let $x_i$ be the production for month $i$.
- Let $o_i$ be the number of items produced by overtime work in month $i$.
- Let $h_i$ and $f_i$ be the number of workers hired and fired, respectively, in the beginning of month $i$.
- Let $s_i$ be the amount of product stored after the end of month $i$.

The constraints then are as follows:

- Production: $x_i = 200w_i + o_i$. The amount produced is based on regular work and overtime work.
- Number of workers: $w_i = w_{i-1} + h_i - f_i, w_i \geq 0$. This constraint reflects how the number of workers may change. There is the natural added constraint that the number of workers be non-negative.
- Surplus: $s_i = s_{i-1} + x_i - d_i \geq 0$. The amount in storage is past surplus plus new production less the demand.
- Overtime limits: $o_i \leq 60w_i$ because a worker can not take on more than 30% overtime.

The objective is to minimize the operational cost:

$$\min \quad 5000\sum_{i=1}^{12} w_i + 2000\sum_{i=1}^{12} h_i + 4000\sum_{i=1}^{12} f_i + 80\sum_{i=1}^{12} s_i + 45\sum_{i=1}^{12} o_i.$$

## 4 Example: Fractional Knapsack

In the weighted knapsack problem, you have a bag that can carry up to $W$ pounds. You have $n$ items labeled $x_1, \ldots, x_n$. Item $x_i$ has weight $w_i$ and value $v_i$. You would like to select items to maximize the sum of the values without exceeding the weight limit of your knapsack.

In the fractional version of this problem, one can select a fraction of an item (reduce the weight) and obtain the same fraction of the value of that item.

The fractional knapsack problem can easily be solved using a greedy algorithm that picks items using the maximum value per pound ordering. This problem can also be encoded as a linear program as follows.

The variables are $f_1, \ldots, f_n$. $f_i$ indicates the fraction of $x_i$ that one selects.

The objective is to maximize the sum of the value:

$$\max \sum_{i=1}^{n} f_i v_i.$$

The constraints are to not exceed the weight capacity of the knapsack and to ensure that $0 \leq f_i \leq 1$.

$$\begin{aligned} \sum_{i=1}^{n} f_i w_i &\leq W; \\ f_i &\geq 0, \quad \forall i; \\ f_i &\leq 1, \quad \forall i. \end{aligned}$$

The linear program described above can be solved in polynomial time (either using the greedy approach or using an interior point methods).

If we insist that $f_i$ is either 0 or 1 then the nature of the problem changes dramatically. We return to the original weighted knapsack problem, which can be solved in pseudopolynomial time using dynamic programming. By insisting that the variables (of interest) be integers we shift the focus to *integer linear programs* (ILPs), and integer linear programs are hard to solve (we do not believe we can solve all integer programs in polynomial time).

## 5 Example: Separating Points

Suppose that we have two sets of points in the plane, the black points $(x_i, y_i), i = 1, \ldots, m$ and the white points $(x_i, y_i), i = m+1, \ldots, m+n$. We wish to separate

them by a straight line $ax + by = c$, so that for all black points $ax + by \leq c$, and for all white points $ax + by \geq c$. In general, this would be impossible. Still, we may want to separate them by a line that minimizes the sum of the "displacement errors" (distance from the boundary) over all misclassified points. Here is the LP that achieves this.

$$\min \; e_1 + e_2 + \cdots + e_m + e_{m+1} + \cdots + e_{m+n}$$

subject to

$$\begin{aligned}
e_1 &\geq ax_1 + by_1 - c; \\
e_2 &\geq ax_2 + by_2 - c; \\
&\vdots \\
e_m &\geq ax_m + by_m - c; \\
e_{m+1} &\geq -(ax_{m+1} + by_{m+1} - c); \\
&\vdots \\
e_{m+n} &\geq -(ax_{m+n} + by_{m+n} - c); \\
e_i &\geq 0, \quad \forall i.
\end{aligned}$$

## 6 Example: Shortest Paths in a Graph

Dijkstra's algorithm is often used to describe a method for finding the shortest path between two vertices in a graph when edge weights are all non-negative. It is possible to formulate the shortest path problem as an LP on a graph $G = (V, E)$. The modelling here makes use of the idea of flow conservation across a vertex.

Let $V = \{v_i\}$ be the set of vertices and $E = \{e_{ij}\}$ be the set of edges. (A vertex is represented as $v_i$ and the edge between vertices $v_i$ and $v_j$ is denoted $e_{ij}$. We can assume that the graph is directed although the formulation is correct for undirected graphs as well.) Let $l_{ij}$ be the length of edge $e_{ij}$. We will use $s$ to denote the source vertex and $t$ to denote the destination vertex. $x_{ij}$ is the indicator variable that is 1 if $e_{ij}$ is on the shortest path between $s$ and $t$ and 0 otherwise.

The LP involves only $x_{ij}$ as the variables. The objective is

$$\min \sum_{e_{ij} \in E} l_{ij} x_{ij}$$

and the constraints are

$$\begin{aligned}
\sum_{v_j \in V} x_{ij} - \sum_{v_j \in V} x_{ji} &= 1, \quad v_i = s; \\
\sum_{v_j \in V} x_{ij} - \sum_{v_j \in V} x_{ji} &= -1, \quad v_i = t; \\
\sum_{v_j \in V} x_{ij} - \sum_{v_j \in V} x_{ji} &= 0, \quad v_i \neq s, v_i \neq t; \\
x_{ij} &\geq 0, \ \forall e_{ij}.
\end{aligned}$$

A key observation here is that this is an LP and not an ILP. We do not need to insist that $x_{ij}$ be an integer. The solution (obtained by the Simplex method or any other method) is guaranteed to have integer values for all variables because any other choice would be suboptimal.

## 7 Example: Network Flow

Suppose we are given a data/traffic network (graph) with edge capacities. The capacity of an edge represents the amount of data/traffic that can be carried on that edge. We would like to know what is the maximum amount of data/traffic that can be supported between a source $s$ and a destination $t$.

For example, consider a graph the represents the road network of Vancouver. Each road can support a certain number of cars per hour. We would like to determine the maximum number of cars per hour that can travel from UBC to downtown Vancouver using all possible routes. This maximum flow measure can then be used to determine if more roads are needed or some roads need expansion.

The maximum network flow problem can also be cast as a linear program. Using notation that is similar to what we used for the shortest path problem we have, we will use $x_{ij}$ to represent the flow along edge $e_{ij}$ and we will use $c_{ij}$ to represent the capacity of edge $e_{ij}$. (We are not considering edge weights in this setting.)

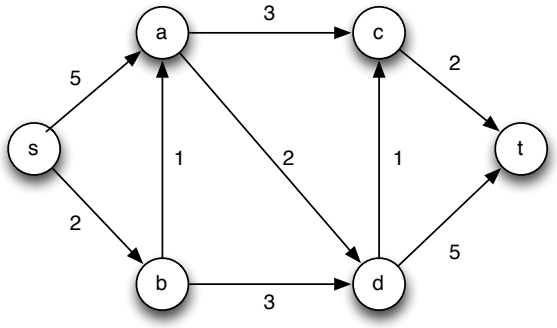


Figure 2: An example network with flow capacities indicated for each edge. The maximum flow between $s$ and $t$ in this network is 6.

Our goal is to maximize the flow from the source (let's denote this amount as $f$) and so the objective is

$$\max\ f$$

subject to the following constraints.

- Capacity constraints: $x_{ij} \leq c_{ij},\ \forall e_{ij} \in E$. The flow along an edge cannot exceed its capacity;
- Flow conservation: $\sum_{v_j \in V} x_{ij} - \sum_{v_j \in V} x_{ji} = 0,\ \forall i \neq s, t$.
- Source constraint: $\sum_{v_j \in V} x_{ij} - \sum_{v_j \in V} x_{ji} = f,\ i = s, \forall j$;
- Sink constraint: $\sum_{v_i \in V} x_{ij} - \sum_{v_j \in V} x_{ji} = f,\ j = t, \forall i$.

## 8 Comments

Linear programs can model a variety of problems. The insight is in transforming a problem into an LP formulation. When a problem can be mapped to an LP formulation then one can rely on standard algorithms (such as the Simplex method) to find a solution. It is, however, not always efficient to use LP formulations. For many problems, where the problem structure is easily understood, other algorithmic techniques (divide-and-conquer, dynamic programming, greedy algorithms) may be more efficient and practical.

When one requires integer solutions in an LP, the problem is an Integer Linear Program (ILP). ILPs may not be easy to solve. In general, ILPs may

take exponential time to solve. In specific instances, one can relax the requirement that solutions be integers, solve the relaxed LP, and then perform some rounding on non-integers to obtain integer solutions. Rounding is applicable to some problems and is not effective for many others.

It is not always possible to map a problem to an LP. This situation occurs when some constraints are non-linear or when the objective function is non-linear. Non-linear programs (NLPs) can be solved by other methods. Some instances of NLPs can be solved in polynomial time (for example, when the constraints and the objective function are convex) but NLPs in general are computationally hard.