Linear Programming

The definitions of LP, and other pieces of the material appear in CLRS Chapter 29

The linear-time algorithm for LP in 2D from MMOM

Slides courtesy of Craig Gotsman
Linear Programming - Example

- Define: (amount amount consumed per day)
  - \(i\) – types of foods (\(1 \leq i \leq d\)).
  - \(j\) – types of vitamins (\(1 \leq j \leq n\)).
  - \(x_i\) – the amount of food of type \(i\) consumed per day).
  - \(a_{ji}\) – the amount of vitamin \(j\) in one unit of food \(i\).
  - \(c_i\) – the number of calories in one unit of food \(i\).
  - \(b_j\) – minimal required amount of vitamin \(j\).

- Constraints (we need to consume some minimal amount of each vitamin):

\[
\begin{align*}
  a_{11}x_1 + a_{12}x_2 + \cdots + a_{1d}x_d & \geq b_1 \\
  \vdots \\
  a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nd}x_d & \geq b_n
\end{align*}
\]

Minimize:

\[
C(x) = c_1x_1 + c_2x_2 + \cdots + c_dx_d
\]
And a less silly example

• Finding maximum flow in a network
• (Linear Programming and ILP are not the only algorithms for solving flow problem. We will see other algorithms. Yet sometimes they are the best options).

• Applications: Find maximum volume of vehicles in a road network, max traffic of messages in a communication network etc etc etc.
Flow networks

Definition. A flow network is a directed graph $G = (V, E)$ with two distinguished vertices: a source $s$ and a sink $t$. Each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v)$. If $(u, v) \notin E$, then $c(u, v) = 0$.

Example:
A flow on a network

Flow conservation
- Flow into $u$ is $2 + 1 = 3$.
- Flow out of $u$ is $0 + 1 + 2 = 3$.

The value of this flow is $1 - 0 + 2 = 3$. 
Flow networks

Definition. A positive flow on $G$ is a function $p : V \times V \to \mathbb{R}$ satisfying the following:

• Capacity constraint: For all $u, v \in V$,
  \[ 0 \leq p(u, v) \leq c(u, v). \]

• Flow conservation: For all $u \in V - \{s, t\}$,
  \[ \sum_{v \in V} p(u, v) - \sum_{v \in V} p(v, u) = 0. \]

The value of a flow (which we want to maximize) is the net flow out of the source:

\[ \sum_{v \in V} p(s, v) - \sum_{v \in V} p(v, s). \]
Network Flow as an LP problem

**Definition.** A *positive flow* on $G$ is a function $p : V \times V \rightarrow \mathbb{R}$ satisfying the following:

- **Capacity constraint:** For all $u, v \in V$,
  $$0 \leq p(u, v) \leq c(u, v).$$

- **Flow conservation:** For all $u \in V - \{s, t\}$,
  $$\sum_{v \in V} p(u, v) - \sum_{v \in V} p(v, u) = 0$$

The *value* of a flow is the net flow out of the source:

$$\max \sum_{v \in V} p(s, v) - \sum_{v \in V} p(v, s)$$

As an LP, the variables are the values $p(u, v)$.

*The capacities $c(u, v)$ are the input.*

There are $n^2$ variables

About $n^2$ constraints (n is number of nodes)
Linear Programming – The Geometry

- Each constraint defines a half-space region in $d$-dimensional space.
- The *feasible region* is the (convex) intersection of these half-spaces.

- We will treat the case $d = 2$, where each constraint defines a *half-plane*.
Degenerate Cases

• The feasible region may be:
  – Empty
  – Unbounded

• The solution may be:
  – Not unique
The Simplex Algorithm

• Assume WLOG that the cost function points “downwards”.
• Construct (some of) the vertices of the feasible region.
• Walk edge by edge downwards until reaching a local minimum (which is also a global minimum).

• In $\mathbb{R}^d$, the number of vertices might be $\Theta(n^{\left\lfloor d/2 \right\rfloor})$. 
LP History

- Mid 20\textsuperscript{th} century: Simplex algorithm, time complexity $\Theta(n^{d/2})$ in the \textbf{worst} case.
- 1980’s (Khachiyan) ellipsoid algorithm with time complexity $\text{poly}(n,d)$.
- 1980’s (Karmakar) interior-point algorithm with time complexity $\text{poly}(n,d)$.
- 1984 (Megiddo) – parametric search algorithm with time complexity $O(C_d n)$ where $C_d$ is a constant dependent only on $d$. E.g. $C_d = 2^{d^2}$.
- The holy grail: An algorithm with complexity independent of $d$.
- In practice the simplex algorithm is used because of its linear \textit{expected} runtime.
O(n^2) Incremental Algorithm

• The idea:
  – Start by intersecting two halfplanes.
  – Add halfplanes one by one and update optimal vertex by solving one-dimensional LP problem on new line if needed.
Incremental Algorithm - Symbols

$h_i$  the $i^{th}$ half plane

$l_i$  the line that defines $h_i$

$C_i$  the feasible region after $i$ constraints

$v_i$  the optimal vertex of $C_i$

Cost function to minimize:  $c(x,y)=y$  
Returns the leftmost point in feasible region
Incremental Algorithm
Basic Theorem

• Theorem:
  1. if $v_{i-1} \in h_i$, then $v_i = v_{i-1}$.  // O(1) check, nothing to do
  2. if $v_{i-1} \notin h_i$, then either
     $C_i = \emptyset$  // terminate
     or
     $C_i = C_{i-1} \cap h_i$ and $v_i$ lies on $l_i$  // run 1D LP

• Proof:
  1. Trivial. Otherwise $v_i$ would not have been optimum before.
Basic Theorem - Cont.

2. Assume that $v_i$ is not on $l_i$. $v_i$ must be in $C_{i-1}$. By convexity, also the segment $v_i v_{i-1}$ is in $C_{i-1}$.

Consider point $v_j$ - the intersection of $v_i v_{i-1}$ with $l_i$. $v_j$ is in both $C_{i-1}$ and $C_i$, and is better than $v_i$.

Contradiction.
Finding $v_i$ given $l_i$

(one-dimensional LP)

- Intersect each $h_j$ ($j<i$) with $l_i$, generating $i-1$ rays representing (unbounded) intervals.
- Intersect the $i-1$ intervals in $O(i)$ time.
- If the intersection is empty then report no solution, else report the lowest point.
Complexity Analysis

\[ T(n) = \sum_{i=3}^{n} O(i) = O(n^2) \]
Incremental Algorithm – $O(n)$
Randomized Version

• Exactly like the deterministic version, only the order of the lines is random.

• **Theorem:** The expected runtime of the random incremental algorithm (over all $n!$ permutations of the input constraints) is $O(n)$. 
Complexity Analysis

- The expected runtime is:

\[
\sum_{i=3}^{n} [O(1)(1 - E(x_i)) + O(i)E(x_i)] \leq O(n) + \sum_{i=3}^{n} [O(i)E(x_i)]
\]

where \( x_i \) is a random variable:

\[
x_i = \begin{cases} 
1 & v_i \neq v_{i-1} \quad \text{// run 1D LP} \\
0 & v_i = v_{i-1} \quad \text{// do nothing}
\end{cases}
\]
Probability Analysis

Backward analysis

- **Question**: When given a solution after $i$ half-planes, what is the probability that the *last* half-plane affected the solution?

- **Answer**: Exactly $2/i$, because a change can occur only if the last half-plane inserted is one of the two half-planes thru $v_i$.
  (Note that $v_i$ depends on the $i$ half-planes, but not on their order)
Complexity Analysis

\[ E(x_i) = \Pr(v_i \neq v_{i-1}) \approx \frac{2}{i} \]

\[ O(n) + \sum_{i=3}^{n} O(i)E(x_i) = O(n) + O\left( \sum_{i=3}^{n} i \cdot \frac{2}{i} \right) = O(n) \]
Just to Make Sure …

• False Claim:
  – The probabilistic analysis is for the average input. Hence there exist bad sets of constraints for which the algorithm’s expected runtime is more than $O(n)$, and there exist good sets of constraints for which the algorithm’s expected runtime is less than $O(n)$.
The maximum-flow problem

**Maximum-flow problem:** Given a flow network $G$, find a flow of maximum value on $G$.

The value of the maximum flow is 4.