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
Example of an LP: The Diet problem

In the diet problem, we will have to compute two values $x$ and $y$, indicating many bananas and oranges we will consume daily, so we get enough many vitamins while minimizing the amount of calories.
Define: (amount consumed per day)
- types of foods: \{oranges, bananas\}
- \( j \) – types of vitamins (\( 1 \leq j \leq n \)).
- \( x \) – number of pounds of oranges we recommend daily
- \( y \) – number of pounds of bananas we recommend daily

// these are the only unknown we have to compute.
- \( a_{ji} \) – the amount of vitamin \( j \) in a unit of food \( i \)
  \( (i=1 \text{ for oranges, } i=2 \text{ for bananas}) \). These are constants.
- \( c_1 \) – the number of calories in an orange.
- \( c_2 \) – the number of calories in a banana.
- \( b_j \) – minimal daily required amount of vitamin \( j \).

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

\[
\begin{align*}
a_{11}x + a_{12}y & \geq b_1 \\
& \vdots \\
a_{n1}x + a_{n2}y & \geq b_n
\end{align*}
\]

Minimize: the total number of calories:

\[
C((x, y)) = c_1x + c_2y
\]
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):

\[
\begin{align*}
\min c^T x \\
\text{Subject to} & \quad Ax \geq b
\end{align*}
\]

Minimize: the total number of calories consumed:

\[
C(x) = c_1 x_1 + c_2 x_2 + \cdots + c_d x_d
\]
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*.
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 $(u, v) \not\in E$, then $c(u, v) = 0$.

**Example:**
Flow networks

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:
\[
\sum_{v \in V} p(s, v) - \sum_{v \in V} p(v, s).
\]
The solution to the linear program is a point in the feasible region that is extreme in the direction of the target function.

**Theorem:** Any bounded linear program that is feasible has a solution, which is a vertex of the feasible region.

**Proof:** Convexity …
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 $\Theta(n^{\lceil d/2 \rceil})$. 
LP History

- Mid 20th century: Simplex algorithm, time complexity $\Theta(n^{d/2})$ in the worst case.
- 1980’s (Khachiyan) ellipsoid algorithm with time complexity poly($n,d$).
- 1980’s (Karmakar) interior-point algorithm with time complexity 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 expected runtime.
**O(n log n) 2D Linear Programming**

- **Input:**
  - $n$ half planes.
  - Cost function that WLOG “points down”.

- **Algorithm:**
  - Partition the $n$ half-planes into two groups.
    - $S$ are all halfplanes contain the point $(0, \infty)$
    - $S'$ all other halfplanes contain the point $(0, -\infty)$
  - Sort them by slopes
  - Compute the upper envelop $U(S)$ and the lower envelop $L(S')$
    (using question from hw1)
  - Scan simultaneously from left to right, and compute intersection of two envelopes - they can intersect only at 2 points (why).
  - Evaluate cost function at each vertex.
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 - Notation

- $h_i$ 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 lowermost point in feasible region
Theorem:

1. If $v_{i-1} \in h_i$, then $v_i = v_{i-1}$. // O(1) check, nothing to do

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

   Consider point $v_j$ - the intersection of $v_iv_{i-1}$ 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:

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

where \( x_i \) is a random variable:

\[
x_i = \begin{cases} 
1 & v_i \neq v_{i-1} \quad // \text{run 1D LP} \\
0 & v_i = v_{i-1} \quad // \text{do nothing}
\end{cases}
\]
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 halfplane inserted is one of the two halfplanes thru \( v_i \). (note that \( v_i \) depends on the \( i \) halfplanes, 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 $O(n)$.

- **True Claim:**
  - The probabilistic analysis is valid for *all* inputs. The expected complexity is over all *permutations* of this input.
Now the input is a collection of half-spaces \( \{h_1 \ldots h_n\} \).

Now \( l_i \) is the plane bounding \( h_i \). (notations are analogous to the 2D case).

We will define \( v_3 \) as the intersection of the planes \( l_1, l_2 \) and \( l_3 \).

We insert the other halfspaces \( \{h_4 \ldots h_n\} \) at a random order, and update \( v_i \) according to the following Theorem:

**Theorem:**

1. if \( v_{i-1} \in h_i \), then \( v_i = v_{i-1} \). // O(1) check,
   nothing to do

2. if \( v_{i-1} \not\in h_i \), then the solution (if exists) is on \( l_i \).
   
   run \( v_i = \text{2DLP}(h_1 \cap l_i, h_2 \cap l_i, h_3 \cap l_i, \ldots, h_{i-1} \cap l_i). \)

Terminates if there is no solution (that is, \( C_i = \emptyset \))
In 3D, the worst case running time is $\Theta(n^3)$ \textit{(prove)}.

However, the expected running time is $O(n)$. In general, the running time in $d$-dimension is $O(d! \ n)$. That is, linear in any fixed (and small) dimension.