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: designing non-fat multi-vitamins diet.

- Define:
  - \( f \) – types of foods (1≤i≤n).
  - \( y_i \) – types of vitamins (1≤j≤m).
  - \( x_{ij} \) – variables (what we want to find). The amount of food of type \( i \) we need to consume.
  - \( a_{ij} \) – 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):
  \[
  a_{ij}x_{ij} + a_{il}x_{il} + \cdots + a_{in}x_{in} \geq b_j
  \]
  \[
  a_{ij}x_{ij} + a_{il}x_{il} + \cdots + a_{in}x_{in} \geq b_j
  \]
- Minimize: the total number of calories consumed:
  \[
  C(x) = c_1x_1 + c_2x_2 + \cdots + c_nx_n
  \]

The solution may be:
- Not unique.

More Geometry

- 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 unique solution, which is a vertex of the feasible region.
- Proof: Convexity ...

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 \( \mathbb{R}^2 \), the number of vertices might be \( O(p^{2/3}) \).
**LP History**
- Mid 20th century: Simplex algorithm, time complexity $O(n^{3+\epsilon})$ in the worst case.
- 1980’s (Khachiyan) ellipsoid algorithm with time complexity poly($n,d$).
- 1980’s (Karmarkar) interior-point algorithm with time complexity poly($n,d$).
- 1984 (Megiddo) – parametric search algorithm with time complexity $O(C_n^d)$ where $C_n$ is a constant dependent only on $d$. E.g. $C_2 = 2^{2^2}$.
- The holy grail: An algorithm with complexity independent of $d$.
- In practice the simplex algorithm is used because of its linear expected runtime. Very very fast in practice. Available on many free and commercial libraries.

**O($n$) Algorithm for LP in 1D**
- Problem: Find the most extreme point on a line, which is in the intersections of half-lines.
- LP: find $x$ such that:
  - $x > a_i$ for $i = 1,2,...,n$ and $x < b_i$.
- Easily solved by checking if $\max\{a_i\} < \min\{b_i\}$
  - Return $\min\{b_i\}$ if yes. Otherwise there is no solution.
- Same idea if the problem is to find a point $(x,y)$ on a given line $l$, and $(x,y)$ maximizes $c_i x + c_j y$ s.t. $a_i x + b_i y < k_i$ for $i = 1,...,n$.

**O($n^2$) Incremental Algorithm for LP in 2D**
- 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**
- $h_j$ the $j^{th}$ half plane.
- $l_j$ the line that defines $h_j$.
- $C_i$ the feasible region after $i$ constraints.
- $v_j$ the optimal vertex of $C_j$.

**Incremental Algorithm Basic Theorem**
- **Theorem:**
  1. If $v_j \not\in h_i$ then $v_j = v_{j-1}$. // O(1) check, nothing to do.
  2. If $v_j \in h_i$, then either $C_i \not\subseteq h_i$. // terminate
     or $\exists C_i = C_i \cap h_i$ and $v_j$ lies on $l_i$. // run 1D LP
- **Proof:**
  1. Trivial. Otherwise $v_j$ would not have been optimum before.

**Basic Theorem - Cont.**
- 2. Assume that $v_j$ is not on $l_j$. $v_j$ must be in $C_{j-1}$.
  - By convexity, also the segment $v_{j-1}v_j$ is in $C_{j-1}$.
  - Consider point $v_{j-1}$ the intersection of $v_{j-1}v_j$ with $l_j$. $v_j$ is in both $C_{j-1}$ and $C_j$ and is better than $v_{j-1}$.
  - Contradiction.
Finding \( v_i \) given \( l_i \) (one-dimensional LP)

- Call the 1D algorithm – (repeated with the new notation):
- Intersect each \( h_j \) 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 \( O(n) \).
- (idea of proof – very similar to the closest pair example).
  - When adding the \( i \)th plane, the probability that we need to solve a 1D LP is small \( 2/i \).
  - With probability \( (i-2)/i \), we need to do almost nothing, since \( v_i = v_{i-1} \).

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)

Finishing the analysis

So in the \( i \)th stage we are spending \( O(i) \) time with probability \((i-2)/i\), and \( O(1) \) time with probability \( 2/i \), so the expected work in this stage is \( O(i \cdot 2(i+1) = O(i) \).

Hence the total expected time is \( O(n) \).

This argument can be done more formal using random variables.

Expected time

Let \( T_i \) denote the expected time spent at stage \( i \). Then \( T_i = 1 \) with probability \((i-2)/i\) and \( T_i = i \) with probability \( 2/i \).

\[
E(T_i) = \sum_{j=1}^{i} j \Pr(T_i = j) = 1 \Pr(T_i = 1) + i \Pr(T_i = i) = 3
\]

Hence the expected total time is

\[
E(\sum_{i=3}^{n} T_i) = \sum_{i=3}^{n} E(T_i) = \sum_{i=3}^{n} 3 = O(n)
\]