There are two kinds of randomized techniques for solving linear programs: random-sampling and randomized incremental algorithms. The first algorithm we will look at is based on random sampling.

First, we introduce a few notations. Let $H$ denote a set of $n$ constraints, and let $\omega(H)$ denote the value of the optimal solution under the constraints in $H$. Consider the following instance of linear programming:

$$\min x_d \quad x = (x_1, x_2, ..., x_d) \in \mathbb{R}^d$$

$$Ax \leq b.$$ 

Each constraint marks a $d$-dimensional half-space where feasible points must lie. The intersection of all $n$ of these half-spaces is a polyhedron, referred to as the *feasible region*. The optimal solution to this problem is a vertex of the convex polytope, which is a point defined by the intersection of the boundary hyperplanes of $d$ of the constraints.

**Definition 1** Let $R$ be a set of constraints. $B \subseteq R$ is called a basis if $\omega(B') < \omega(B) \quad \forall B' \subset B$. This means removing any constraint from $B$ would cause the optimal objective function value to decrease. $B$ is a basis for $R$, defined by $B(R)$, if $\omega(B) = \omega(H)$ and $B$ is a basis.

**Claim 1** For a set $H$ of constraints and an infeasible solution $x \in \mathbb{R}^d$, let $V \subset H = \{h \mid x \text{ violates } h\}$. Then $V \cap B(H) \neq \emptyset$.

**Definition 2** A constraint $h$ is enforcing if $\omega(G/\{h\}) < \omega(G)$.

### 4.1 Random-Sampling Based Algorithms

The algorithm *Sample_LP*, described in Figure 4.1, builds up a set of constraints by adding a random sample of the constraints at each step which is both small and must contain at least one new enforcing constraint, if the step is ”successful” (i.e., line (*) is executed). Since there are $d$ constraints in the basis, there can be at most $d$ successful iterations of this algorithm.

**Claim 2** The probability that an iteration is successful is at least $1/2$. 


Sample\_LP(H)

\[
\text{if } |H| \leq 9d^2 \text{ then } \\
\quad \text{return } \text{Simplex}(H) \\
\text{end if } \\
V \leftarrow H, S \leftarrow \emptyset \\
\text{while } V \neq \emptyset \text{ do } \\
\quad \text{choose } R \subseteq H/S \text{ random subset of size } r = 9d^2 \\
\quad x \leftarrow \text{Sample\_LP}(R \cup S) \text{ (**)} \\
\quad V \leftarrow \{h \in H \mid x \text{ violates } h\} \\
\quad \text{if } |V| \leq 2d\sqrt{n} \text{ then } \\
\quad \quad S \leftarrow S \cup V \text{ (⋆)} \\
\quad \text{end if } \\
\text{end while } \\
\text{return } x
\]

Figure 4.1: Sample\_LP algorithm

WeightedSample\_LP(H)

\[
\text{if } |H| \leq 9d^2 \text{ then } \\
\quad \text{return } \text{Simplex}(H) \\
\text{end if } \\
\text{for all } h \in H \text{ do } \\
\quad \mu(h) = 1 \\
\text{end for } \\
\text{repeat } \\
\quad \text{choose } R \subseteq H/S \text{ random subset of size } r = 9d^2 \\
\quad (\text{Pr}\{h \text{ is selected}\} = \mu(h)/\mu(H)) \\
\quad x \leftarrow \text{Simplex}(R) \\
\quad V \leftarrow \{h \in H \mid x \text{ violates } h\} \\
\quad \text{if } \mu(V) \leq 2\mu(H)/9d \text{ then } \\
\quad \quad \text{for all } h \in H \text{ do } \\
\quad \quad \quad \mu(h) \leftarrow 2\mu(h) \text{ (⋆)} \\
\quad \quad \text{end for } \\
\quad \text{end if } \\
\text{end repeat } \\
\text{until } V = \emptyset \\
\text{return } x
\]

Figure 4.2: WeightedSample\_LP
When we call the simplex algorithm for \( R \cup S \), \( |R \cup S| \leq 3d\sqrt{n} \). Let \( T(n) \) be the maximum expected running time of the algorithm with \( n \) constraints. The recurrence for this algorithm is

\[
T(n) \leq 2dT(3d\sqrt{n}) + O(nd)
\]

Its solution is \( T(n) = O(d^d n) \).

We can improve the running time of this algorithm by using the iterative reweighting technique instead of trying to discover the basis one constraint at a time. In the algorithm \( WeightedSample_{LP} \), when the random sample is good, i.e., line (\( * \)) is executed, we double the weight of each of the violated constraints. The constraints in \( B(H) \) should quickly receive very high weights as a result of being repeatedly selected and violated. The algorithm is described in Figure 4.2.

As before, we call an iteration successful if \( * \) is executed. The probability of a successful iteration is still at least \( 1/2 \), but the number of successful iterations is now slightly higher.

**Claim 3** The number of successful iterations is \( O(d \log n) \).

**Proof:** We will show that the total weight of \( B(H) \) grows faster than the total weight of \( H \), so that if \( V \neq \emptyset \) after \( d \log n \) iterations,

\[
\sum_{h \in B(H)} \mu(h) > \sum_{h \in H} \mu(h)
\]

which is a contradiction.

Because \( V \) must always contain at least one constraint from \( B(H) \), at least one of the basis constraints will double in weight after each iteration of the loop. This means after \( kd \) iterations,

\[
\sum_{h \in B(H)} \mu(h) = \sum_{h \in B(H)} 2^{n_h}
\]

with \( n_h \) being the number of times \( h \) appeared in \( V \). Since one of the basis constraints appears at each iteration, \( \sum_{h \in B(H)} n_h \geq kd \), which implies \( \sum_{h \in B(H)} \mu(h) \geq d2^k \).

Initially \( \mu(H) = n \). After a successful iteration, the weight \( \mu(H) \) increases by a factor of \((1 + 2/9d)\). Therefore, after \( kd \) successful iterations,

\[
\sum_{h \in B(H)} \mu(h) \leq d2^k \leq \mu(H) \leq n \left(1 + \frac{2}{9d}\right)^{kd} \leq n \exp(2k/9)
\]

Hence \( k = O(d \log n) \). □

The expected running time of \( WeightedSample_{LP} \) is then

\[
O(d \log n)(d^{d/2+O(1)} + dn) \approx O(d^2 n \log n + d^{d/2+O(1)} \log n)
\]

By invoking \( WeightedSample_{LP} \) in step (\( ** \)) of \( Sample_{LP} \), we obtain a new randomized algorithm for \( LP \) with expected running time

\[
d^2 n + O(d^3 \sqrt{n} \log n + d^{d/2+O(1)} \log n)
\]
4.2 Randomized Incremental Algorithms

The two algorithms above relied on the use of random sampling to run the simplex algorithm on a problem with a smaller number of constraints. We will now study a randomized incremental algorithm which takes a different approach. The idea is to randomly choose a constraint and throw it out if it is redundant. Otherwise, we can add it to the basis.

We again assume the problem is non-degenerate, so that the basis size is $d$. The goal of the algorithm $\text{Incr}_LP(H)$ is to return a basis for $H$. It will do this by selecting a random constraint, and recursing on $H/\{h\}$. If the basis returned, $B$, does not violate $h$, it is returned. Otherwise, $h$ must be a part of the basis, and we can reduce the dimension of the problem by restricting to the hyperplane described by $h$. The algorithm is described in detail in Figure 4.3.

Let $T(n, d)$ denote the expected running time of $\text{Incr}_LP$ in $d$ dimensions with $n$ constraints. Since step $(\ast)$ is executed if $h \in B(H)$, the probability that $(\ast)$ is executed is $d/n$. Hence, we obtain the following recurrence:

$$T(n, d) \leq T(n-1, d) + d + \frac{d}{n}[dn + T(n-1, d-1)]$$

Claim 4 $T(n, d) \leq b(d+1)!n$

Proof: We will show this by induction.

For $d \geq 2, b \geq 4, d^2 - bd! < 0$ Hence,

$$T(n, d) \leq b(d+1)!n$$

The expected running time of this algorithm could be improved if we take advantage of the fact that $B$ probably provides some information for the $(d-1)$-dimensional problem instead of throwing $B$ away and solving the reduced dimension problem from scratch. A new algorithm which takes advantage of this is...
\[
\text{Basis\textunderscore LP}(G, T)
\]

\[
\begin{align*}
\text{if } G = T & \text{ then} \\
& \text{return } G \\
\text{end if}
\end{align*}
\]

\[
\begin{align*}
h & \leftarrow \text{random constraint from } G/T \\
B & \leftarrow \text{Basis\textunderscore LP}(G/\{h\}, T) \\
\text{if } B & \text{ does not violate } h \text{ then} \\
& \text{return } B \\
\text{else} \\
& \text{return } \text{Basis\textunderscore LP}(G, \text{Basis}(B \cup \{h\}))
\end{align*}
\]

---

**Figure 4.4:** Basis\textunderscore LP

The function \(\text{Basis}(G)\) used in this algorithm returns a basis for the set of \(d + 1\) constraints passed to it, \(d\) of which form a basis without the new constraint \(h\). The algorithm computes the intersection of \(h\) with each size \(d - 1\) subset of the other constraints and evaluate the objective function at each of the \(d\) points to find the new basis.

The running time of this algorithm can be bounded by bounding the probability that \(B\) violates \(h\). If the size of \(G\) is \(i\), the probability is \(\leq d/(i - |T|)\), because there are at most \(d\) constraints in \(B(G)\) and \(h\) is chosen uniformly at random from \(G/T\). However, we can get a tighter bound on this probability by introducing the notion of hidden dimension.

**Definition 3** A constraint \(h \in G\) is called enforcing in \((G, T)\) if \(\omega(G/\{h\}) < \omega(T)\).

If all \(d\) constraints in \(T\) are enforcing in \((G, T)\) then \(T = B(G)\).

**Definition 4** Let \(\Delta_{G,T} = d - |\{h \in T \mid h \text{ is enforcing in } (G, T)\}|\). Then we will call \(\Delta_{G,T}\) the hidden dimension of \((G, T)\). It represents the number of basis constraints we haven’t found yet.

We can now bound the probability that \(B\) violates \(h\) by \(\Delta_{G,T}/(i - |T|)\) and argue that the hidden dimension decreases by at least one at every iteration of the algorithm; in fact, it can be argued that the hidden dimension decreases much faster.

Suppose \(g_1, g_2, \ldots, g_l\) are enforcing constraints in \((G, T)\), numbered such that \(\omega(G/\{g_1\}) < \omega(G/\{g_2\}) < \ldots < \omega(G/\{g_n\})\).

**Claim 5** \(g_1, g_2, \ldots, g_l\) are enforcing constraints in \((G, \text{Basis}(T \cup \{g_l\}))\).

So when \(h = g_l\), all of \(\{g_1, g_2, \ldots, g_l\}\) will be enforcing for \((G, \text{Basis}(T \cup \{g_l\}))\). This means the hidden dimension in the recursive call reduces by \(l\). Any of the \(g_i\) is equally likely to be \(h\). Let \(T(n, i)\) be the
expected number of iterations for $|G| = n$ and $\Delta_{G,T} = i$. The recurrence for the algorithm is

$$T(n, i) = T(n - 1, i) + \frac{1}{i} \sum_{j=1}^{i} i[T(n - 1, j) + dn]$$

$$T(n) = O(d^4 2^d n)$$

However a more complicated analysis can be used to show that this algorithm actually runs in time $O(e^{O(\sqrt{\log d})} n)$. The missing details can be found in the paper by Matousek, Sharir, and Welzl.