IMPLEMENTACION DEL METODO PRIMAL-DUAL PARA PROGRAMACION LINEAL

Trabajo de licenciatura de: **Mónica L. Ingratta**
Dirigido por: **Lic. Irene Loiseau**
Depto. de Computación, Universidad de Buenos Aires

Abril, 1996

Agradecimientos

Quiero agradecer especialmente a la Lic. Irene Loiseau, por su infinita paciencia y buena voluntad para conmigo; a Mauricio Resende, por su buena disposición al responder mis consultas; a mi familia, mi novio y amigos por saber entender mi falta de disponibilidad y tiempo para ellos.

Introducción

En este trabajo se describe la implementación de un algoritmo para resolver el problema de programación lineal

$$
\begin{aligned}
(P) \qquad & \min c^T x \\
& \text{sa } Ax = b \\
& \quad x \geq 0
\end{aligned}
$$

donde $c \in R^n$, $b \in R^m$, $A \in R^{mxn}$, $n > m$.

El problema de programación lineal fue resuelto por primera vez por George B. Dantzig en 1947 [Dan]. Dantzig desarrolló el método Simplex, que aún hoy es el más utilizado para resolver esta clase de problemas. Sin embargo, y si bien en la práctica es eficiente, no tiene complejidad polinomial[1]. Esto es una desventaja, pues implica que con el Simplex no se puede resolver en tiempo razonable una serie de problemas. Un ejemplo de ellos se puede ver en [KlM].

En un principio se creía que esta característica era intrínseca al problema de programación lineal, hasta que en 1979 L. Khachiyan publicó el primer algoritmo polinomial [Kha]. En realidad, hubo uno anterior, presentado por Dikin [Dik], pero no tuvo trascendencia.

Desde entonces, han surgido varios algoritmos polinomiales para el problema de programación lineal, entre los cuales ha tenido mayor resonancia el de Karmarkar [Kar]. El implementado en este trabajo es uno de ellos: se trata del primal-dual, desarrollado por Clovis C. Gonzaga [Gon]. Forma parte de la familia de algoritmos de punto interior (así denominados en oposición al Simplex -que en cada iteración obtiene vértices de la región de factibilidad-, pues se mueven en el interior de dicha región).

---

[1] La complejidad es un criterio utilizado para medir la eficiencia de un algoritmo, en base al tamaño de los datos del problema, *n*. Se dice que un algoritmo tiene complejidad polinomial si la cantidad de iteraciones que requiere para resolver el problema está acotada por una función polinomial del tamaño del mismo *P(n)*. [Edm]

1. Conceptos fundamentales

1.1. El problema de programación lineal

Se desea resolver el problema de programación lineal:

(P) $\min c^T x$

$\text{sa } Ax = b$

$x \geq 0$

donde $c \in R^n$, $b \in R^m$, $A \in R^{mxn}$, $n > m$.

Se supone la matriz A de rango completo, y la región de factibilidad

$$S = \{x \in R^n \mid Ax = b, x \geq 0\}$$

es acotada y tiene un interior

$$S^o = \{x \in R^n \mid Ax = b, x > 0\}$$

no vacío.

Todo problema (P) tiene asociado su **problema dual**, que se define como:

(D) $\max b^T y$

$\text{sa } A^T y + z = c$

donde $y \in R^m$, $z \in R^n$, $z \geq 0$.

Las variables z se denominan **slacks duales**.

Un problema de programación lineal y su dual están estrechamente ligados. También lo están sus soluciones (si las hay); en particular, tienen el mismo valor óptimo.

De esta relación, surge el concepto de **gap de dualidad**:

Sea $x \in S$; sea $y \in R^m$ el punto dual factible asociado. Se llama gap de dualidad a:

$\Delta = c^T x - b^T y$.

Es claro que en caso de optimalidad, el gap de dualidad es nulo, pues si x^* e y^* son las soluciones óptimas del primal y del dual, respectivamente, el gap es:

$\Delta = c^T x^* - b^T y^* = 0$

El siguiente lema propone otra forma de calcular el gap de dualidad, que nos servirá más adelante.

Lema 1.1: Dado el par (x, z) donde $x \in S$, y (y,z) es dual-factible para algún $y \in R^m$. Entonces, el gap de dualidad es $\Delta = x^T z$.

Dem:

$\Delta = c^T x - b^T y = c^T x - (Ax)^T y = c^T x - x^T A^T y$

De (D) se desprende que: $A^T y = c - z$. Luego,

$\Delta = c^T x - x^T (c - z) = c^T x - x^T c + x^T z = x^T z$. ♦

Esto nos resultará útil para saber si una solución calculada es óptima. Pero antes de ver cómo, necesitamos definir más conceptos:

Asociados a la transformación lineal que representa A se hallan dos subespacios de R^n:

- el **espacio nulo** $\mathcal{N}(A) = \{ x \in R^n \mid Ax = 0 \}$
- el **espacio rango** $\mathcal{R}(A^T) = \{ x \in R^n \mid \exists w \in R^m, x = A^T w \}$.

Estos espacios son ortogonales, pues:

dados $x \in \mathcal{N}(A)$, $y \in \mathcal{R}(A^T)$, por definición de espacio rango $\exists w$ tal que

$y = A^T w$. Luego

$x^T y = x^T A^T w = (Ax)^T w = 0w = 0$.

Entonces, cualquier vector $v \in R^n$ puede ser descompuesto como

$v = vp + vp'$, donde vp es la proyección de v en $\mathcal{N}(A)$ y vp' es la proyección de v en $\mathcal{R}(A^T)$.

Dado que el operador de proyección es lineal, puede ser representado por una matriz P, tal que $vp = Pv$. Su complemento ortogonal es
$vp' = v - vp = v - Pv = (I - P)v$.

Siendo A de rango completo, se puede demostrar que la matriz de proyección es
$P_A = I - A^T(AA^T)^{-1}A$.

La proyección simplifica muchos cálculos, y nos ayuda a encontrar las slacks duales que necesitamos para calcular el gap de dualidad, según lo muestra el siguiente lema.

Lema 1.2: Un punto $z \varepsilon R^n$ es una slack dual factible para (D) si y sólo si $z \geq 0$ y $P_A z = P_A c$.

Dem:
Sea $z \geq 0$. z es una slack dual factible si y sólo si $\exists w \varepsilon R^m$ tal que
$A^T w + z = c \quad <=> \quad c-z = A^T w$
Pero c-z se puede descomponer unívocamente en:
$c-z = P_A(c-z) + P_A'(c-z) = P_A(c-z) + A^T w$
Pero c-z ya está en el espacio rango, por lo cual $P_A(c-z) = 0$
$<=> \quad P_A c - P_A z = 0 \quad <=> \quad P_A z = P_A c$ ♦

Otra herramienta de gran utilidad es la **transformación de escala**, que consiste en:
Sean $x \varepsilon R^n$, $y \varepsilon R^n$, y $D \varepsilon R^{nxn}$ una matriz diagonal positiva. Entonces una transformación de escala en el problema (P) es un cambio de variables $x = Dy$.
Dado un punto x^o, escalar alrededor de x^o es aplicar la transformación de escala $x = X_o y$, donde $X_o = diag(x^o_1, ..., x^o_n)$.

Si se aplica la transformación de escala al problema (P), se obtiene un nuevo problema equivalente, tal que:
* la función objetivo es:
$$c^T x = c^T X_o y = c^T X_o^T y = (X_o c)^T y.$$
pues X_o es una matriz diagonal y entonces $X_o = X_o^T$.
* la región de factibilidad es:
$$Ax = b \quad <=> \quad AX_o y = b.$$
* la condición de no-negatividad de las variables se mantiene, pues la matriz X_o tiene todos sus elementos positivos. Luego,
$$x \geq 0 \quad <=> \quad X_o y \geq 0 \quad <=> \quad y \geq 0.$$
Por lo tanto, el problema de programación lineal escalado alrededor de x^o es:
(PE) $\min c^T y$
sa $Ay = b$
$y \geq 0$
donde $A = AX_o$, $c = X_o c$.

Veamos cómo afecta la transformación de escala a las soluciones factibles para (D).

Lema 1.3: Sean $y \varepsilon R^m$, $z \varepsilon R^n$, $z \geq 0$, $X \varepsilon R^{nxn}$ una matriz diagonal. Entonces, (y, z) es una solución dual factible de (P) si y sólo si
(y, Xz) es una solución dual factible de (PE).

Dem:
(y, z) es una solución dual factible de (P) si y sólo si
$A^T y + z = c$, con $z \geq 0$.
Premultiplicando por X,
$XA^T y + Xz = Xc \quad <=> \quad (AX)^T y + Xz = Xc \quad <=>$
$A^T y + Xz = c$
lo cual representa una solución dual factible de (PE). ♦

1.2 Un primer algoritmo de resolución

En esta sección se plantea una primera forma de resolver el problema de programación lineal: el algoritmo de máximo descenso escalado.

Dado el problema de optimización general
$\min \{ f(x) \mid x \in S^o \}$
y un punto $x^o \in S^o$,
la **dirección de máximo descenso** es:
$\min \{ \nabla f(x^o)^T d \mid ||d|| \leq \varepsilon, d \in \mathcal{N}(A) \}$.
Es decir que la dirección de máximo descenso es aquélla que realiza el mínimo de la aproximación lineal de la función f alrededor de x^o, en una bola centrada en dicho punto.

Este problema fue estudiado por Cauchy el siglo pasado. Su solución surge a partir de la desigualdad de Cauchy-Schwarz, y es múltiplo de
$h = -P_A \nabla f(x^o)$

La dirección de máximo descenso provee una solución fácil para problemas de minimización sin restricciones. Sin embargo, en el caso de un problema restringido, puede ser sumamente ineficiente, ya que al minimizar la función en una bola, no se tiene en cuenta la región de factibilidad. Para poder tenerla en cuenta, se debe minimizar la función en otra clase de región, preferentemente de forma similar a la de factibilidad. La más sencilla entre éstas es un elipsoide con ejes paralelos a los ejes coordenados. Se puede entonces escalar el problema alrededor de x^o para transformar el elipsoide en una bola centrada en e (el vector unitario) y proyectar el gradiente escalado para obtener el mínimo en la región.

De esta forma, se deduce el siguiente algoritmo para resolver un problema de minimización utilizando el concepto de dirección de máximo descenso:

Algoritmo de máximo descenso escalado (scaled-steepest descent)
$k := 0$
Repetir
Aplicar escala: $A := AX_k$; $g := X_k \nabla f(x^k)$
Calcular dirección: $h := -P_A g$
Hallar el tamaño del paso: $y := e + \lambda h, \ y > 0$
Aplicar escala: $x^{k+1} := X_k y$
$k := k + 1$
Hasta *convergencia*

La transformación de escala transporta x^k a e. Para hallar el tamaño del paso, se realiza una búsqueda lineal que por lo general consiste en una minimización aproximada de f en la dirección *h*. El parámetro λ determina el tamaño del paso que se da en la dirección *h*, y se toma de forma tal de hacer el nuevo punto positivo.

Si se aplica un algoritmo de tales características al problema (P) se obtiene el método *afín-escala*, propuesto por Dikin en 1967, método que puede ser visto como una variante simplificada del algoritmo de Karmarkar [Dik, Bar, VMF, AKR].

El algoritmo afín-escala requiere puntos interiores para generar "buenas" regiones de minimización, lo cual se consigue controlando el tamaño del paso. Aquí se distinguen diversas propuestas: Dikin utilizaba paso de longitud uno en la dirección de búsqueda, es decir, $\lambda = ||h||^{-1}$; sin embargo, esto no resulta eficiente. Otros autores proponen utilizar un porcentaje (fijo, superior al 95%) del máximo paso posible en el cuadrante

positivo; este enfoque evita la frontera, pero los puntos hallados pueden llegar a acumularse cerca de ella.

Una forma de evitar la frontera y mantenerse alejado de ella es definir un centro para el politopo S, y considerar simultáneamente los dos objetivos: reducción del costo y centralización. Este concepto es común a todos los algoritmos de punto interior.

Al elegir el centro, se debe tener en cuenta cuán fácil es su cálculo. El primer centro que nos viene a la mente es el de gravedad. Sin embargo, calcularlo implica resolver un problema más costoso que el de programación lineal... Otra alternativa es el centro de un elipsoide de máximo volumen inscripto en S (propuesto por Khachiyan [KhT]), pero su cálculo también es demasiado trabajoso. En contraste, se puede calcular más fácilmente el *centro analítico*, propuesto por Sonnevend [Son]:

Dado el politopo S, se llama **centro analítico** al punto
$\chi = \text{argmin} \{ p(x) \mid x = (x_1, \ldots, x_n) \in S^o \}$
donde p es la función barrera: $p(x) = - \Sigma \log(x_i)$.
(Recordemos que (P) exige la positividad de las variables.)
La función barrera tiene derivadas
$\nabla p(x) = - x^{-1} \quad \nabla p(e) = - e$
$\nabla^2 p(x) = X^{-2} \quad \nabla^2 p(e) = I$
Dado que $\nabla^2 p(x)$ es definida positiva en S^o, p es estrictamente convexa. Además de ello, p crece indefinidamente al acercarse a la frontera de S, por lo que el centro analítico está bien definido.

Este centro tiene además una propiedad altamente deseable, que se desprende del siguiente lema.

Lema 1.4: Las transformaciones de escala no afectan a las variaciones de la función barrera.

Dem:
Sea una matriz diagonal positiva D, y sean dos puntos u y v $\in S^o$.
$p(Du) = - \Sigma \log(d_{ii} u_i) = - \Sigma [\log(d_{ii}) + \log(u_i)] = - \Sigma \log(d_{ii}) - \Sigma \log(u_i) = p(u) - \Sigma \log(d_{ii})$
Luego
$p(Du) - p(Dv) = p(u) - \Sigma \log(d_{ii}) - [p(v) - \Sigma \log(d_{ii})] = p(u) - p(v)$ ♦

El aplicar transformaciones de escala no sólo es inocuo -pues no afecta a las variaciones de la función barrera- sino que además da lugar a derivadas muy sencillas ($\nabla p(e) = -e$, $\nabla^2 p(e) = I$) y simplifica bastante las expresiones matemáticas.

Veamos un último concepto necesario para presentar el algoritmo primal-dual.

1.3 La función penalizada

En cada iteración, y dado un punto en S^o, se desea obtener un punto *mejor* en dos sentidos: de menor costo y más lejano de la frontera. Es decir, se tienen dos objetivos, por lo que es conveniente tomar una combinación de ambos. Dicha combinación se puede encarar de diversas maneras; por ejemplo, la **función penalizada**

$$f_\alpha(x) = \alpha c^T x + p(x), \quad \text{con } \alpha > 0$$

[Fri, FMC]. En esta función compuesta, el parámetro α determina el peso relativo de ambos términos entre sí: a mayor α, mayor importancia se da al costo y menor énfasis se pone en evitar la frontera.

El gradiente de la función penalizada escalada en e es:
$\nabla f_\alpha(e) = \alpha c - e$

y la dirección de máximo descenso en e es una combinación de los vectores $P_A c$ (denominado dirección de reducción de costo) y $-P_A e$ (dirección de centrado).

Aquí surge otra razón para proyectar: en e las derivadas son muy sencillas y además se da la siguiente particularidad.

Lema 1.5: Sea la función penalizada f_α para algún α fijo, no negativo. Entonces la dirección de máximo descenso coincide con la dirección de Newton-Raphson.

Dem:
Sin pérdida de generalidad, asumamos que x = e. Entonces la dirección de máximo descenso es:
$h(e, \alpha) = -\alpha P_A c + P_A e$
El desarrollo de Taylor de la función penalizada alrededor de e es:
$\nabla f_\alpha(e+h) \cong \nabla f_\alpha(e) + \nabla^2 f_\alpha(e)\, h = \nabla f_\alpha(e) + Ih$
La dirección de Newton-Raphson corresponde a aquella que realiza el mínimo de dicha aproximación, proyectada.
$P_A \nabla f_\alpha(e) + h(e, \alpha) = 0$
$=> h(e, \alpha) = -P_A \nabla f_\alpha(e) = -\alpha P_A c + P_A e$

Pero la dirección de Newton-Raphson es invariante frente a la escala, no depende de la métrica utilizada, luego para cualquier punto $x \in S^o$, vale que:
$h(x, \alpha) = -P_A \nabla f_\alpha(x)$ ♦

Dado que para la función penalizada, el algoritmo de máximo descenso escalado coincide con el método de Newton-Raphson, se puede deducir que los dos son igualmente eficientes para determinar el centro analítico.

2. El método primal-dual

En esta sección se describe en general una familia de algoritmos de punto interior, y en particular el primal-dual.

2.1 Algoritmos *path-following*

Dada la función penalizada $f_\alpha(x) = \alpha c^T x + p(x)$, con $\alpha \varepsilon R^+$, se denomina **punto central** asociado a α a:
$x(\alpha) = \text{argmin} \{ f_\alpha(x) \mid x \varepsilon S^o \}$
Un punto central $x(\alpha)$ está bien definido puesto que f_α es estrictamente convexa y crece indefinidamente cerca de la frontera de S. Luego para cada α, $x(\alpha)$ es único, y se deduce de:
$P_A \nabla f_\alpha(x) = 0 \quad <=> \quad \alpha P_A c - P_A x^{-1} = 0$

Se define entonces el **camino central** (*central path*) como la curva determinada por
$\alpha \varepsilon R^+ -> x(\alpha) \varepsilon S^o$

El camino central es una curva suave, y tiene una propiedad muy deseable: a medida que α crece, más se acerca a la solución óptima de (P). Fue estudiado por primera vez por Bayer y Lagarias [BaL], y en profundidad por Megiddo [Meg].

Los algoritmos que se mueven a lo largo del camino central para hallar soluciones de un problema de programación lineal reciben el nombre de ***path-following***, y pertenecen a la categoría de algoritmos de punto interior. El primero fue presentado por Renegar [Ren]. Todos los *path-following* responden al siguiente esquema conceptual:

Algoritmo path-following conceptual
Sea $\alpha \varepsilon (w^-, w^+)$ una parametrización del camino central, y sea $x^o := x(\alpha^o)$.
$k := 0$
Repetir
    Elegir $\alpha^{k+1} > \alpha^k$
    Llamar a un algoritmo de minimización interno para hallar $x^{k+1} := x(\alpha^{k+1})$
    $k := k+1$
Hasta *convergencia*

Lamentablemente, hallar un punto central es un problema de minimización no lineal que no puede ser resuelto exactamente en tiempo razonable, por lo que se debe trabajar con puntos cercanos al camino central. Ello da lugar al siguiente algoritmo:

Algoritmo path-following implementable
Sea $x^o = x(\alpha^o)$.
$k := 0$
Repetir
    Elegir $\alpha^{k+1} > \alpha^k$
    Llamar a un algoritmo de minimización interno para hallar x^{k+1} cercano a $x(\alpha^{k+1})$, partiendo de x^k
    $k := k+1$
Hasta *convergencia*

Para definir un algoritmo particular en base al general, es necesario determinar los siguientes elementos:
¤ un algoritmo de minimización interno. Por lo general, es el de máximo descenso escalado o Newton-Raphson.
¤ un criterio de proximidad al camino central, dado que no es posible calcular de forma exacta y en tiempo razonable un punto central. Lo deberá aplicar el algoritmo de

minimización interno para decidir que el punto hallado es cuasi-central (cercano a un punto central). Este problema fue tratado por varios autores [Pol, MHM, Ans].

¤ un criterio de parada, es decir, cuándo se considera que se alcanzó la convergencia. Normalmente se basa en el gap de dualidad -recordar que éste es nulo en caso de optimalidad.

¤ cómo actualizar el parámetro α en dada iteración, de forma tal que acelere la convergencia.

Como ya mencionamos, es habitual tomar un criterio de convergencia basado en el gap de dualidad. Por ello, el lema continuación y su corolario brindan una forma de calcular una slack dual factible asociada a un punto interior y el gap de dualidad.

Lema 2.1: Sea $\alpha \varepsilon R^+$ y un punto $x \varepsilon S^o$. Si $h(x,\alpha) < e$, entonces

$$z = X^{-1} \frac{(e - h(x,\alpha))}{\alpha}$$

es una slack dual factible, y el gap de dualidad es:

$$\Delta = \frac{(n - e^T h(x,\alpha))}{\alpha} \leq \frac{(n + \delta(x,\alpha)\sqrt{n})}{\alpha}$$

Dem:

Por hipótesis, $z > 0$

Según el lema 1.3, z es factible para (P) si y sólo si

$$z = Xz = X X^{-1} \frac{(e - h(x,\alpha))}{\alpha} = \frac{e - h(x,\alpha)}{\alpha}$$

es factible para (PE). Proyectando z,

$$P_A z = \frac{P_A e - P_A h(x,\alpha)}{\alpha}$$

Pero como $h(x,\alpha) \varepsilon \mathcal{N}(A)$, $\quad P_A h(x,\alpha) = h(x,\alpha) = -\alpha P_A c + P_A e$

Entonces,

$$P_A z = \frac{P_A e + \alpha P_A c - P_A e}{\alpha} = \frac{\alpha P_A c}{\alpha} = P_A c$$

Por lo tanto, según el lema 1.2, z es dual factible para (PE) y por consiguiente z es dual factible para (P).

El gap de dualidad es

$$\Delta = x^T z = x^T X^{-1} \frac{(e - h(x,\alpha))}{\alpha} = e^T \frac{e - h(x,\alpha)}{\alpha} = \frac{n - e^T h(x,\alpha)}{\alpha}$$

Pero $-e^T h(x,\alpha) \leq ||e||.||h(x,\alpha)|| = \sqrt{n}\,\delta(x,\alpha)$

$$\therefore \Delta = \frac{n - e^T h(x,\alpha)}{\alpha} \leq \frac{n + \sqrt{n}\,\delta(x,\alpha)}{\alpha} \quad \blacklozenge$$

Corolario 2.1: Dado $\alpha \varepsilon R^+$ y un punto $x \varepsilon S^o$. Si $\delta(x,\alpha) < 1$, valen las conclusiones del lema 2.1.

Dem:

Si $\delta(x,\alpha) < 1$, entonces

$\forall i, |h_i(x,\alpha)| < 1$.

Luego, $h(x,\alpha) < e$ y por lo tanto se dan las condiciones del lema 2.1.♦

2.2 El algoritmo primal-dual

Este algoritmo propuesto por Gonzaga en [Gon] es el path-following basado en la función penalizada [Fri] [FMC],

$f_\alpha(x) = \alpha c^T x + p(x)$

Debe su denominación a la estrecha relación existente entre el gap de dualidad y la parametrización del camino central, descripta más adelante.

Veamos los elementos que lo distinguen de sus pares:

2.2.1 Criterio de proximidad

Se utilizó un criterio sugerido por Polak en [Pol], que versa:
Sea $\varepsilon_k > 0$, tal que ε_k tiende a 0 a medida que k crece. Entonces un punto x es cuasi-central si y sólo si $|| P\nabla f_\alpha(x) || < \varepsilon_k$.
Así expresado no es satisfactorio pues depende de la escala empleada, pero aplicando una transformación de escala se salva el inconveniente.

Se tomó entonces como medida de cercanía al camino central a la longitud del paso de máximo descenso escalado (o Newton-Raphson), es decir,
$\delta(x,\alpha) = ||h(x,\alpha)||$
donde, tal como en las secciones anteriores, $h(x,\alpha)$ es la dirección de máximo descenso en el espacio escalado.

2.2.2 Criterio de parada
Se puede asumir sin pérdida de generalidad que el problema (P) está planteado en término de datos enteros. Se define entonces el tamaño del problema como
$L = \ell + n + 1$
donde ℓ es la cantidad de bits necesarios para A, b, c; n es la cantidad de variables del problema (incluyendo las slacks).

2^L es un número grande, mayor que el resultado de cualquier cálculo aritmético usando datos que no se repiten; también es mayor que cualquier determinante menor de A. Por lo tanto, 2^{-L} es un número muy pequeño, con las siguientes propiedades:

i) Si x es vértice de S, ninguna componente de x está en el intervalo $(0, 2^{-L})$.
Dem: sea A_B una submatriz no singular de A, de dimensión mxm. Entonces
$x_B = A_B^{-1}b = (\text{cofactor}(A_B))^T b / \det A_B > (\text{cofactor}(A_B))^T b / 2^L$
Como A y b están compuestos por enteros, y $0 \leq x$ es un vértice,
$(\text{cofactor}(A_B))^T b = 0$ o
$(\text{cofactor}(A_B))^T b \in N$
Luego, $x_B = 0$ o $x_B > 1 / 2^L = 2^{-L}$.

ii) Si x es vértice de S, $c^Tx - c^Tx^* = 0$ o $c^Tx - c^Tx^* > 2^{-L}$
Esto se desprende de i) y de la integralidad de los costos.

Por lo tanto, el criterio de parada es $\Delta < 2^{-L}$

El problema ahora es calcular el gap de dualidad, para lo cual nos valemos del siguiente lema.

Lema 2.2: Sea $x(\alpha)$ el punto central asociado a un $\alpha \in R^+$. Entonces,
$z(\alpha) = \dfrac{(x(\alpha))^{-1}}{\alpha}$
es una slack dual factible y el gap de dualidad asociado a $x(\alpha)$ y $z(\alpha)$ es
$\Delta = n / \alpha$

Dem:
Según lo visto anteriormente, un punto central es tal que:
$\alpha P_A c - P_A x(\alpha)^{-1} = 0 \quad <=> \quad P_A x(\alpha)^{-1} = \alpha P_A c$
$<=> \quad \dfrac{P_A x(\alpha)^{-1}}{\alpha} = P_a c$
Siendo $x(\alpha) > 0$, $x(\alpha)^{-1} / \alpha > 0$. Luego satisface las condiciones necesarias y suficientes para ser una slack dual factible según el lema 1.2. Entonces el gap de dualidad es:

$\Delta = x(\alpha)^T z(\alpha) = x(\alpha)^T (x(\alpha))^{-1} / \alpha = n / \alpha$ ♦

El valor del gap de dualidad es entonces constante en el camino central, para esta parametrización. He aquí el origen del nombre del algoritmo.

2.2.3 Actualización del parámetro

En cada iteración se pretende reducir el gap de dualidad en un cierto porcentaje $\beta \in (0, 1)$. Es decir, se busca β tal que:
$\Delta^{k+1} = \beta\Delta^k = \beta n / \alpha^k$
Pero $\Delta^{k+1} = n / \alpha^{k+1}$. Luego
$\beta n / \alpha^k = n / \alpha^{k+1}$
$\therefore \alpha^{k+1} = \alpha^k / \beta$

2.2.4 Complejidad del algoritmo

Al comienzo del trabajo dijimos que los algoritmos de punto interior tienen complejidad polinomial con respecto a la cantidad de variables involucradas (n). Veamos en particular la complejidad del primal-dual.

La cantidad de iteraciones del algoritmo principal (en contraposición al algoritmo de minimización interno) depende del tamaño del problema, de la siguiente forma:

Lema 2.3: Asumiendo que el algoritmo comienza con un gap de dualidad $\Delta_0 < 2^L$, la cantidad de iteraciones es del orden de $O(L/\log(1/\beta))$

Dem:
Si todavía no se alcanzó la convergencia, en la iteración k,
$\Delta^{k+1} = \beta\Delta^k$
$\Rightarrow 2^{-L} \leq \Delta^{k+1} \leq \beta^k \Delta_0 \leq \beta^k 2^L$
Aplicando logaritmo en base 2, resulta
$-L \leq k.\log\beta + L$
$\Rightarrow -2L \leq k.\log\beta$
$\Rightarrow k \leq -2L / \log\beta = 2L/\log(1/\beta)$ ♦

Podemos afirmar entonces que la complejidad depende de la elección de β.
Si se toma independiente de n, la complejidad es O(L). Sin embargo, la experiencia indica que es posible que el algoritmo de máximo descenso escalado requiera una gran cantidad de iteraciones para hallar el punto cuasi-central.
Por el contrario, si se toma β en función de n, tal que $1/\beta = 1 + \nu/\sqrt{n}$, la cantidad de iteraciones del algoritmo interno depende de ν, y la complejidad del algoritmo general es $O(\nu\sqrt{n}L)$. Para valores pequeños de ν, sólo es necesaria una iteración del algoritmo interno [Ren].

El algoritmo de máximo descenso escalado tiene la complejidad de la inversión de una matriz, es decir, $O(n^3)$, con lo cual la complejidad total es $O(n^{3.5}L)$. Karmarkar [Kar] propuso una forma de cálculo que la reduce a $O(n^3L)$.

Las demostraciones de estas afirmaciones se pueden consultar en la bibliografía referenciada, pues exceden al objetivo de este trabajo.

Algoritmo primal-dual

Input:

A = matriz de restricciones
x = punto inicial
c = vector de costos
b = vector de recursos
L = tamaño del problema
α = parámetro
itmax = número máximo de iteraciones
dist = distancia mínima entre dos soluciones para no ser consideradas iguales.

Output:

x = solución óptima

Algoritmo:

Calcular la norma del vector de costos	$\|\|c\|\|$
MIENTRAS (gap $> 2^{-L}$ Y k $<$ itmax)	
Calcular el gradiente escalado	$g := (\alpha c^T x / \|\|c\| - x; Mx_a)$
Calcular la matriz de proyección escalada	$P := I - (AX)^T (AX^2A^T)^{-1} (AX)$
Calcular la dirección de máximo descenso	$h := -XPg$
Calcular el nuevo punto	$x^{k+1} := x^k + \lambda h,\ x^{k+1} > 0$
SI ($\|x^{k+1} - x^k\| <$ dist)	
TERMINAR: x es un punto fijo	
SINO	
Actualizar el parámetro	$\alpha := (1 + 0.95/\sqrt{n})\alpha$
Calcular el gap de dualidad	$gap := n / \alpha$
FIN	
k := k+1	
FIN	
SI (gap $< 2^{-L}$)	
DEVOLVER x	
FIN	

3 Implementación del método primal-dual

En esta sección se cubren los aspectos prácticos de la implementación del método primal-dual.

3.1 Elección del parámetro α

El parámetro α determina el peso relativo entre la función objetivo original y la función de penalización, por lo cual es de esperar que sea un numero entre 0 y 1. En la experiencia se observó que tomar mayores valores de α no siempre aceleraba la convergencia, sino todo lo contrario: en ocasiones el algoritmo diverge -y rápidamente- al tomar valores cercanos a 1 y converge con valores inferiores. En las experiencias realizadas, 0.5 resultó ser un buen valor para considerar.

3.2 Resolución del problema

En cada iteración, el mayor esfuerzo computacional radica en el cálculo de la matriz de proyección. Esto implica la resolución de un sistema lineal de la forma
$(AA^T)y = A$
Dado que la matriz AA^T es simétrica definida positiva, se utilizó el método de Cholesky, que consiste en factorizar la matriz del sistema como el producto de una matriz triangular inferior y su traspuesta. Es decir, si se desea resolver el sistema $Pz = b$ según el método de Cholesky, con $P \in R^{mxm}$, se debe calcular la matriz $L \in R^{mxm}$ tal que $LL^T = P$. Una vez obtenida L, se resuelve el sistema:

$$Lv = b$$
$$L^Tz = v$$

equivalente al dado. La resolución de estos nuevos sistemas es inmediata, dado que L es triangular.

Dadas las matrices $A \in R^{mxn}$ y $X \in R^{nxn}$, en cada iteración del algoritmo primal-dual se debe resolver el sistema:
$(AA^T)\, y = A \quad <=> \quad (AX^2A^T)y = AX.$
En cada iteración la matriz del sistema es numéricamente distinta, pero su estructura es invariante. Por lo tanto, la estructura de la matriz factor L también es invariante, dado que depende exclusivamente de la estructura de la matriz del sistema. Por consiguiente, se pueden diferenciar los siguientes pasos para resolver el sistema:
. cálculo de la estructura de AX^2A^T.
. cálculo de la estructura de L, o *factorización simbólica*.
. cálculo de los elementos de AX^2A^T.
. cálculo de los elementos de L, o *factorización numérica*.

Cabe destacar que los dos primeros pasos se pueden realizar sin tener en cuenta los valores numéricos de los elementos de las matrices A, X y L, por lo que basta con hacerlos una sola vez, al comienzo de la ejecución del algoritmo primal-dual. Ambos pasos constituyen la llamada *fase analítica*.

En cambio, los últimos dos pasos deben ser repetidos en cada iteración del algoritmo primal-dual, para actualizar los valores de AX^2A^T y L de acuerdo al nuevo punto x calculado. Estos pasos constituyen la llamada *fase numérica*, y responden a la estructura calculada en la fase analítica.

3.2.1 Cálculo de la estructura de AX^2A^T

Previo a la primera iteración, se analiza la estructura no nula de la matriz, es decir, se estudia qué elementos de $M = AX^2A^T$ son distintos de cero. Se intenta de esta forma reducir el tiempo y esfuerzo de procesamiento en cada iteración.

Sea m_{ij} un elemento de M.
$=>\ m_{ij} = \Sigma q_{is} r_{sj} = \Sigma(a_{is} x_{ss})(x_{ss} a_{js})$
Entonces los elementos no nulos serán aquellos m_{ij} tales que para algún s, $a_{is} \neq 0$ y $a_{js} \neq 0$, puesto que x_{ss} nunca será nulo. Luego, para determinar la estructura de la matriz M, se recorren las columnas de la matriz A: por cada par de elementos no nulos de una columna cualquiera (a_{is} y a_{js} con $i<j$), el elemento m_{ji} es no nulo.

3.2.2 Factorización simbólica

El método de Cholesky requiere calcular la matriz triangular inferior $L = (l_{ij})$ de forma tal que
$LL^T = AX^2A^T$
L es a lo sumo tan rala como M, es decir, tiene elementos nulos en posiciones en que M los tiene[2], pero puede tener valores no nulos en posiciones en que M tiene ceros:
$l_{jj}^2 = m_{jj} - \Sigma_{i<j} l_{ji}^2$
$l_{ij} = (m_{ij} - \Sigma_{s<j} l_{is} l_{js}) / l_{jj} \quad \forall i > j$

Cuando L tiene elementos no nulos en posiciones en las que M tiene ceros, se dice que hubo **fill-in**. Evidentemente lo ideal es que el fill-in sea mínimo, puesto que es deseable conservar la raleza lo más posible: mientras más elementos no nulos haya, más pesado es el cálculo y más lenta la resolución.

Una forma de minimizar el fill-in es permutar las filas y columnas de la matriz, y factorizar la matriz permutada. La importancia de esta permutación se ilustra con el siguiente ejemplo:

$$M = \begin{matrix} x & x & x & x & x \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{matrix} \quad => \quad L = \begin{matrix} x & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ x & x & x & x & 0 \\ x & x & x & x & x \end{matrix}$$

(Una x indica un elemento no nulo.)

Una matriz con sólo una columna densa dio lugar a un factor completamente denso. Sin embargo, si se permutan la primera fila con la última, y la primera columna con la última:

$$M_P = \begin{matrix} x & 0 & 0 & 0 & x \\ 0 & x & 0 & 0 & x \\ 0 & 0 & x & 0 & x \\ 0 & 0 & 0 & x & x \\ x & x & x & x & x \end{matrix}$$

el factor resulta

$$L_P = \begin{matrix} x & 0 & 0 & 0 & 0 \\ 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \\ x & x & x & x & x \end{matrix}$$

¡No hubo fill-in!

Desafortunadamente, encontrar la permutación que da lugar al mínimo fill-in es un problema NP-completo, por lo que se debe recurrir a alguna heurística que la aproxime. En este trabajo se ha utilizado la heurística de ordenamiento por grado mínimo (*minimum degree ordering*), desarrollada con mayor detalle en la sección 3.2.5.

---

[2]Excepto por elementos numéricamente nulos (aquellos que se hubieran anulado al hacer los cálculos).

Cabe señalar que, si bien al calcular esta permutación se está extendiendo el proceso, sólo se realiza una vez, al comienzo, con lo cual no es tanto lo que se agrega, frente a la ventaja de trabajar con una matriz factor más pequeña y manejable.

3.2.3 Permutación de la matriz de proyección

En cada iteración se debe calcular la matriz de proyección dada por:
$I - (AX)^T (AX^2A^T)^{-1} AX$
Sea P la matriz de permutación que representa el ordenamiento calculado por la heurística. Luego PAP^T es la matriz del sistema con sus filas y columnas permutadas.
Se desea resolver el sistema $AX^2A^Ty = AX$.
Este equivale a:
$AP^TPX^2P^TPA^TP^TPy = AX$
pues $P^T = P^{-1}$
$<=> (PAP^T)(PX^2P^T)(PA^TP^T)Py = PAX$
$<=> (PAP^T)(PX^2P^T)(PAP^T)^TPy = PAX$
Se resuelve entonces el sistema equivalente:
$$(PAP^T)(PX^2P^T)(PAP^T)^TPy = PAX$$
y por lo tanto la matriz que se factoriza es
$$(PAP^T)(PX^2P^T)(PAP^T)^T$$
que da lugar a un fill-in menor.

De esta forma, se obtiene la solución del sistema pemutada. Para obtener la solución correctamente ordenada se le debería aplicar la permutación inversa.

La matriz de proyección se calcula como
$I - (AX)^T (AX^2A^T)^{-1} AX = I - (AX)^TP^Ty = I - (PAX)^Ty$
Pero entonces es posible trabajar con la matriz del sistema permutada, disminuyendo así el procesamiento y el fill-in simultáneamente.

3.2.4 Detalles de implementación

La intención de esta sección es comentar cómo se encaró la implementación del método, utilizando los elementos teóricos mencionados anteriormente.

3.2.4.1 Estructuras de datos

Los problemas a resolver se suponen planteados de la forma standard
$\min c^Tx$
sa $Ax = b$
$x > 0$

Considerando que a menudo las matrices de estos sistemas son ralas, es posible reducir la cantidad de memoria utilizada. Típicamente esto se intenta utilizando una estructura de datos particular, consistente en:
a) un vector en el que cada elemento indica la posición inicial de cada columna (IA)
b) otro vector con los índices de fila de cada elemento no nulo (JA)
c) un tercer vector que contiene el valor de los elementos no nulos (A)
(suponiendo la matriz almacenada por columnas)
De esta forma se evita el desperdicio de almacenamiento de los ceros [AKV].

Por ejemplo, la matriz

$$M = \begin{matrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 7 & 0 & 9 \end{matrix}$$

se representaría mediante la siguiente estructura de datos:
IA = (0; 2; 4; 7)
JA = (0; 2; 0; 1; 0; 1; 2)
A = (1; 7; 2; 5; 3; 6; 9)

Obviamente el uso de esta estructura no es gratis: a cambio requiere el cálculo de cada dirección. Sin embargo, sigue siendo preferible, puesto que el hecho de usar las matrices completas demanda tanta memoria que se corre el riesgo de no poder aplicar el algoritmo en problemas medianos o grandes (depende del equipo). Además se estarían tomando recursos que podrían necesitar otros usuarios concurrentes.

Una estructura similar se empleó para la matriz factor L, con el agregado de la "compresión" de índices. Esto es, si se detecta que una columna repite los índices de fila de una anterior, en lugar de almacenarlos repetidos, se guarda un puntero al inicio de la segunda columna con respecto a la primera.

Por ejemplo, la matriz

$$L = \begin{matrix} 1 & 0 & 0 & 0 \\ 4 & 5 & 0 & 0 \\ 7 & 0 & 9 & 0 \\ 8 & 3 & 4 & 2 \end{matrix}$$

daría lugar al vector de índices de fila:
SUB = (0, 1, 2, 3, 1, 3)
y al vector de inicio de columnas en SUB:
ISUB = (0, 4, 2, 3)

3.2.4.2 Positividad de las variables

El problema de programación lineal exige en todo momento la positividad de las variables. Debemos entonces asegurar que así sea en cada iteración. Por lo tanto, al calcular un nuevo punto no podemos limitarnos a calcular la dirección de máximo descenso, sino que necesitamos asegurar que al dar un paso en esa dirección no nos salgamos de la frontera.

Es decir, dado x^k positivo, calculamos
$x^{k+1} = x^k(1 + \lambda h) > 0$
$<=> 1 + \lambda h > 0$ pues el punto anterior es positivo
$<=> \forall i, \lambda_i h_i + 1 > 0.$

En esta implementación se tomó $\forall i,\ \lambda_i = (1+||h||)^{-1}$. Veamos que cumple con la condición requerida:
$|| h || = \max |h_i| => ||h|| + 1 > ||h|| \geq -h_i\ \forall i$, y $1 + ||h|| > 0$
Luego,
$1 > -h_i (1+||h||)^{-1} <=> 1 + h_i (1+||h||)^{-1} > 0$

Por lo tanto
$x^{k+1} = x^k(1 + \lambda h) = x^k[1 + h(1+||h||)^{-1}] > 0$

3.2.5 Heurística de ordenamiento por grado mínimo

Tal como mencionamos anteriormente, es conveniente minimizar el fill-in para reducir el procesamiento. Para ello, usamos la heurística de ordenamiento por grado mínimo, que es el objeto de esta sección.

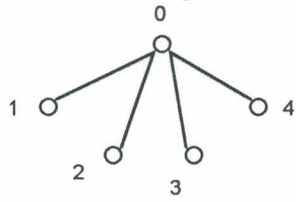
Esta heurística surge de considerar a la matriz del sistema (simétrica) como la matriz de incidencia de un grafo, donde cada variable representa a un nodo y un elemento no nulo

en la posición (i,j) indica que existe un arco entre los nodos i y j (es decir, i y j son adyacentes).

Por ejemplo, la matriz:

$$M = \begin{matrix} x & x & x & x & x \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{matrix}$$

representa al grafo:



A fin de ver cómo reducir el fill-in, estudiamos la estructura no-nula de la matriz original y la factor. El cálculo de la columna k-sima de la factor depende tanto de la matriz original como de las columnas 1 a k-1 de la factor:

Dado el grafo equivalente a la matriz M, sea el nodo k, adyacente a los nodos i y j (tal que $i>j>k$). Esto significa que $m_{ik}\neq 0$ y $m_{jk}\neq 0$, con lo cual

$l_{ik} = (m_{ik} - \Sigma l_{is}l_{ks})/l_{kk} \neq 0$

(ya que no se tienen en cuenta los elementos numéricamente nulos).

Análogamente, $l_{jk} \neq 0$.

Luego, $l_{ik} * l_{jk} \neq 0$, y por ende

$l_{ij} = (m_{ij} - \Sigma l_{is}l_{js})/l_{jj} \neq 0$.

Por lo tanto, los nodos i y j son adyacentes en L.

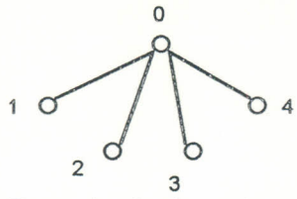
Entonces:

- los elementos no nulos de la columna k-sima de la matriz factor son los nodos adyacentes al k en el grafo. Notar que si i y j son nodos adyacentes a un tercero, pero $a_{ij}=0$ -o sea, i y j no son adyacentes en A- l_{ij} es un elemento de fill-in.
- la influencia de la columna k-sima en las columnas posteriores se traduce en el grafo eliminando el nodo k y uniendo todos los nodos adyacentes al k entre sí. Es decir que el conjunto de ejes añadidos en cada paso de eliminación corresponde al conjunto de fill-in. Resulta entonces que minimizar el fill-in equivale a minimizar la cantidad de arcos añadidos al eliminar un nodo.

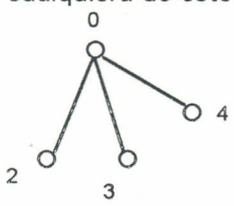
La **heurística de ordenamiento por grado mínimo** consiste en numerar los nodos según la cantidad de nodos adyacentes, y eliminarlos siguiendo esa secuencia [GeL]. Ello explica su nombre: se denomina **grado** de un nodo al cardinal de su adyacencia.
Entonces, el ordenamiento de los nodos según su grado determina la permutación que se aplicará a la matriz original antes de factorizarla, o lo que es lo mismo, el orden en que se calcularán las columnas de la matriz factor.

Ejemplo:
Sea la matriz simétrica del ejemplo anterior

$$M = \begin{matrix} x & x & x & x & x \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{matrix}$$

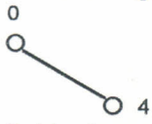


El grado de 0 es 4, mientras que para todos los demás el grado es 1. Si se elimina cualquiera de estos últimos, por ejemplo el 1, el grafo resultante es:



Ahora el grado de 0 es 3, y el de los demás sigue siendo 1. Si se elimina por ejemplo el 2, resulta el grafo:

El mínimo grado lo tienen 4 ó 3. Eligiendo el 3, queda:



Evidentemente, cualquiera de los dos nodos, 0 ó 4, realiza el mínimo, por lo que es indistinto tomar uno o el otro.
Luego, la permutación calculada es: 1, 2, 3, 0, 4.

En este ejemplo en particular la heurística generó una permutación óptima, puesto que no hubo fill-in. Sin embargo, no siempre el resultado es óptimo; en este caso fue así debido a que el grafo original era un árbol.

Tal como se vio en el ejemplo, en cada paso puede haber más de un nodo que realice el mínimo grado. En esta implementación se tomó el de mínimo índice.

3.2.5.1 Más allá del concepto

Hemos visto que la eliminación simétrica de Gauss o Cholesky se puede interpretar como una sucesión de transformaciones de un grafo, o una sucesión de grafos de eliminación. Esto ha sido ampliamente estudiado por Parter [Par] y Rose [Ros].

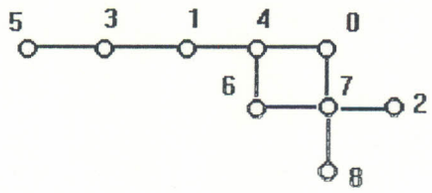
A fin de facilitar la implementación, es deseable representar las distintas transformaciones al grafo en términos del grafo original. Una forma de hacerlo es mediante el concepto de conjuntos de alcance, descripto a continuación.

Dado un conjunto de nodos S, y un nodo $x \notin S$. Se dice que el nodo x es alcanzable desde otro nodo y a través de S si existe un camino $(y, v_0, \ldots, v_p, x)$ de y a x, con $v_i \varepsilon S$ (p puede ser nulo). Se define entonces el **conjunto de alcance** de un nodo y a través de S como:
$Reach(y, S) = \{x \notin S \ / \ x$ es alcanzable desde y a través de S $\}$

Es claro que si $S_i = \{x_0, \ldots, x_{i-1}\}$, todo nodo x_j alcanzable desde x_i a través de S_i, constituye un nodo adyacente a x_i en el paso i-simo. Pero entonces el conjunto de

alcance Reach(x_i, {x_0, ..., x_{i-1}}) representa los índices de fila de la columna i-sima de L. De esta forma, se puede caracterizar el conjunto de fill-in como:
F = {{x_i, x_j} / x_j ε Reach(x_i, {x_0, ..., x_{i-1}}}

Por ejemplo, en el grafo



sea S = { 0, 1, 2, 3}
=> Reach(4, S) = { 5, 6, 7}

El grafo equivale a la matriz

```
        x
        o x
        o o x
        o x o x
 M =    x x o o x
        o o o x o x
        o o o o x o x
        x o x o o o x x
        o o o o o o o x x
```

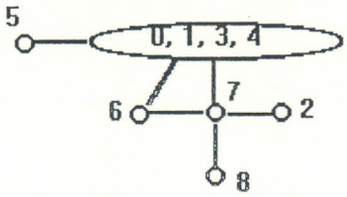
(dado que M es simétrica, basta con mostrar la estructura bajo la diagonal.)
y la estructura de no-ceros queda como sigue:

```
        x
        o x
        o o x
        o x o x
 L =    x x o * x
        o o o x * x
        o o o o x * x
        x o x o * * x x
        o o o o o o o x x
```

(los asteriscos representan elementos de fill-in)
Los elementos de la columna 4 son precisamente los correspondientes a las filas 5, 6, 7, coincidiendo con el conjunto de alcance de 4 calculado anteriormente.

Veamos entonces cómo representar la eliminación según la estructura de la matriz original. Sea G_i el grafo de eliminación obtenido al eliminar el nodo x_i de G_{i-1}. G_i se genera mediante los siguientes pasos:
1) determinar el conjunto de adyacencia $Ady_{Gi-1}(x_i)$ en G_{i-1}
2) Eliminar el nodo x_i y su adyacencia de la estructura de adyacencia
3) Para cada nodo y ε $AdyG_{i-1}(x_i)$, se calcula el nuevo conjunto de adyacencia de y en G_i como la unión de $Ady_{Gi-1}(y)$-{x_i} U $Ady_{Gi-1}(x_i)$ U {y}
El quid es cómo almacenar la estructura de adyacencia: si se guarda toda la estructura explícitamente, puede requerir demasiada memoria, incluso más que para el grafo original. Si en cambio, la estructura se guarda implícitamente, se torna muy pesado el cálculo en cada paso (cada vez hay que recalcular la estructura). Resulta entonces conveniente una solución intermedia, lograda reduciendo los caminos a los nodos no eliminados del grafo. Es decir, se fusionan nodos eliminados conectados entre sí, formando "supernodos". Esto no altera la estructura de adyacencia, pero sí acorta los caminos, con lo cual se reduce la cantidad de memoria necesaria para su almacenamiento.

Aplicando esto al ejemplo anterior, daría lugar al nuevo grafo para S = {0, 1, 2, 3, 4}:



Resulta entonces el algoritmo de ordenamiento por mínimo grado:

S := 0

Grado(x) := | Ady(x) | $\forall x \varepsilon X$

Repetir

Elegir y ε X-S tal que Grado(y) = min Grado(x)

Numerar el nodo y

T := S U {y}

Actualizar el grado Grado(u) := Reach(u, T) $\forall u \varepsilon$ X-S

S := T

hasta que S = X

Notar que $\text{Reach}(u, T) = \begin{cases} \text{Reach}(u, S) & \text{si } u \notin \text{Reach}(y, S) \\ \text{Reach}(u, S) \cup \text{Reach}(y, S) - (u, y) \end{cases}$

Este algoritmo fue el que se implementó en este trabajo, mediante las rutinas:

¤ESTRU: calcula la permutación mediante la heurística de ordenamiento por mínimo grado.

Se vale de las demás rutinas presentadas a continuación.

¤ ALCANCE: determina el conjunto de alcance de un nodo dado.

¤ ACTGRADO: recalcula el grado de un nodo ante una eliminación.

¤ UNIR: forma los supernodos.

¤ TRANSFO: transforma la estructura de adyacencia ante una eliminación.

¤ FSIMB: realiza la factorización simbólica.

Todas éstas fueron desarrolladas en Borland C++.

3.3 Obtención de un punto inicial

Para poder utilizar el método primal-dual, se debe contar con un punto inicial que sea interior y factible. Dado que por lo general no se dispone de un punto semejante[3], es necesario calcularlo.

Una manera de hacerlo es resolver el problema de programación lineal en dos etapas o fases: en la fase uno calcular un punto interior factible, que pueda ser usado como punto inicial del algoritmo primal-dual, la fase dos. [AKR] [Mar] [PRV]

Entonces, la fase uno trata de hallar un punto factible interior al problema de programación lineal dado. Es decir, dado:

$$\begin{aligned} &\min c^T x \\ &\text{sa } Ax = b \\ &\quad 0 \le x \end{aligned}$$

con $c, x \in R^n$; $b \in R^m$; $A \in R^{mxn}$,
hallar un punto x^o tal que:

$$\begin{aligned} Ax^o &= b \\ x^o &> 0 \end{aligned}$$

Para ello, se resuelve el problema::

$$\begin{aligned} &\min Mx_a + \omega c^T x \\ &\text{sa } Ax + vx_a = b \\ &\quad 0 \le x \end{aligned}$$

con $M, \omega \in R$, $x_a \in R$, $v \in R^m$, $0 \le \omega$, $0 \le x_a$.

Se llama a éste problema **aumentado** por cuanto se forma añadiendo al problema original una variable (x_a) con su respectiva columna en la matriz de restricciones (v). Esta variable recibe el nombre de **artificial**, y en cada iteración representa la infactibilidad del punto x^k.

Luego, partiendo de un punto inicial x^o cualquiera, se calcula la variable artificial, su costo y su correspondiente columna en la matriz de restricciones de modo tal que el punto (x^o, x_a) sea factible. En cada iteración, es aconsejable recalcular la columna de infactibilidad, además de la variable, a fin de evitar excesivos errores de redondeos.
De esta forma se resuelve el problema aumentado.

3.3.1 Elección del parámetro ω

El parámetro ω determina el peso de la función objetivo original en el problema aumentado. Para todo problema de programación lineal existe un valor ω' tal que todo problema aumentado con peso $0 < \omega < \omega'$ tiene la misma solución que el problema original. Sin embargo, determinar ω' implica resolver un problema no lineal de igual tamaño que el original.

Para $\omega = 0$, la fase uno es estrictamente una fase de *factibilidad*, esto es, constiste en resolver el problema de obtener un punto factible. Dicho punto es luego utilizado para la fase dos, o fase de *optimalidad*, en la cual se resuelve el problema original en busca de la solución óptima. En este caso, si al finalizar la fase uno la variable artificial fuera positiva, se puede afirmar que la región de factibilidad es vacía; en caso contrario, se procede a la fase dos.

---

[2]El método primal-dual no admite un punto inicial nulo, puesto que en cada paso el nuevo punto se calcula como $x^{k+1} = x^k + hx^k$, siendo h la dirección de máximo descenso. Luego, el 0 es un punto fijo; si se tomara $x^o = 0$, en cada paso el nuevo punto obtenido sería el mismo 0, sin poder salir de él.

Para $\omega>0$, el problema aumentado tiene una función objetivo compuesta: tiene en cuenta tanto la variable artificial como la función objetivo original. En este caso, si al finalizar la fase uno la variable artificial es nula, el punto obtenido no sólo es factible sino que es el óptimo. En cambio, si la función objetivo del problema aumentado es no acotada, pero la variable artificial es nula en todos los puntos de la dirección factible no acotada, se puede afirmar que el problema original es no acotado. Por último, si la fase uno finaliza con la variable artificial positiva, entonces o la región de factibilidad es vacía o el peso tomado fue demasiado grande.

3.3.2 Elección de un punto inicial para la fase uno

Así como la fase dos requiere un punto inicial interior, la fase uno admite un punto inicial cualquiera, cuyo único requisito es la positividad (evidentemente, debe ser interior para la formulación de la fase uno, pero no para el problema original). Sin embargo, el algoritmo es fuertemente dependiente del punto inicial, en el sentido que para un punto inicial "malo" puede divergir, o ser demasiado lenta la convergencia. A modo de ejemplo, citamos el problema de prueba SHARE2B, para el cual ejecutamos exactamente el mismo programa para dos puntos iniciales distintos: en un caso alcanzó la convergencia en 10 iteraciones; en el otro, 160...

El criterio utilizado en esta implementación para calcular un punto inicial se inspiró en el propuesto en [PRV]. Consiste en lo siguiente:
Sean γ , $\rho \in R^+$. Entonces,

$$x_i = \begin{cases} \gamma[0.5 + \dfrac{\rho||c||}{c_i} - \sqrt{0.25 + \left(\dfrac{\rho||c||}{c_i}\right)^2}] & \text{si } c_i>0 \\ \gamma[0.5 + \dfrac{\rho||c||}{c_i} + \sqrt{0.25 + \left(\dfrac{\rho||c||}{c_i}\right)^2}] & \text{si } c_i<0 \\ 0.5\,\gamma & \text{si } c_i=0 \end{cases}$$

Evidentemente, esto no incluye a la variable artificial, que se inicializa con la norma de la columna añadida.

En los problemas de prueba, resultó un buen valor para ρ 0.2, mientras que fue necesario variar γ (para muchos casos, se tomó el valor 0.5).

3.3.3 Algoritmo de la fase uno

El algoritmo utilizado en la fase uno está basado en el método primal-dual, con consideraciones ad-hoc:
¤ Trabaja con la matriz de restricciones extendida con la columna correspondiente a la variable artificial, la columna de infactibilidad. Dicha columna se recalcula en cada iteración como:
$v = \dfrac{(b-Ax)}{||b-Ax||}$,
y la variable artificial es $x_a = ||b-Ax||$

¤ Cambia el criterio de parada. El método primal-dual original finaliza cuando el gap de dualidad se hace más pequeño que una cierta cota. Sin embargo, este criterio no sirve para la fase uno, porque el hecho de partir de un punto cualquiera hace que la convergencia sea mucho más lenta. Luego se establecieron las siguientes condiciones de parada:
a. Que la variable artificial se haga despreciable (menor que una cierta tolerancia). En este único caso, se considera que la fase uno terminó exitosamente.
b. Que la cantidad de iteraciones supere un máximo preestablecido.

c. Que la distancia entre el nuevo punto calculado y el anterior sea despreciable, es decir, que se haya caído en un punto fijo.

¤ Cambia la función objetivo. Además de la variable artificial, se incorpora una función barrera para evitar la frontera, con un peso que varía según el problema a resolver. En suma, la función a minimizar resulta:

$$f = \frac{c^T x}{||c||} - K \Sigma \log(x_i) + M x_a$$

donde $M = 10^p |c^T x^0| / ||x_a||$, con p = 0 ó 1, y

$K = 10^q ||c||$, con q = 1 ó 5, o

$K = 10 |c^T x|$

según el problema (estos valores surgieron empíricamente).

Algoritmo para obtener un punto inicial (fase uno)

Input:

A = matriz de restricciones
x = punto inicial
c = vector de costos
b = vector de recursos
inftol = tolerancia de infactibilidad
w = peso de la función objetivo original
itmax = cantidad máxima de iteraciones
dist = distancia mínima entre dos soluciones para no ser consideradas iguales.

Output:

x = punto inicial para el método primal-dual.

Algoritmo:

Generar un punto inicial	x
Calcular la norma del vector de costos	$\|\|c\|\|$
Calcular la columna de infactibilidad	$v := b - Ax$
Calcular la norma de la columna de infactibilidad	$\|\|v\|\| := \max \|v_i\|$
Calcular el costo de la columna de infactibilidad	$M := \|c^T x\| / \|v\|$

MIENTRAS ($k < itmax$ Y $|v| > inftol$)

Inicializar la variable artificial	$x_a := \|\|v\|\|$
Normalizar la columna de infactibilidad	$v := v / \|\|v\|\|$
Calcular el gradiente escalado	$g := (\omega c^T x / \|\|c\|\| - Kx ; Mx_a)$
Calcular la matriz de proyección escalada	$P := I - (AX)^T (AX^2A^T)^{-1}(AX)$
Calcular la dirección de máximo descenso	$h := -Pg$
Calcular la norma de la dirección	$\|\|h\|\|$
Calcular el nuevo punto	$x^{k+1} := x^k (1 + h\lambda)$

SI ($|x^{k+1} - x^k| < dist$)

TERMINAR: x es un punto fijo

SINO

Calcular la columna de infactibilidad	$v := b - Ax$
Calcular la norma de la columna de infactibilidad	

FIN

$k := k + 1$

FIN

SI ($|v| < inftol$)

DEVOLVER x

FIN

4 Prueba del método primal-dual

Para probar esta implementación del método primal-dual, se han utilizado algunos de los problemas de NETLIB. Se trata de un conjunto standard de problemas de programación lineal, que se pueden recuperar a través de Internet en la dirección electrónica de AT&T (`netlib.att.com`, directorio `NETLIB/LP/DATA`). Mayor información acerca de NETLIB se puede obtener en [Gay] o [DoG].

Todos estos problemas están expresados en formato MPS [Sch], por lo cual fue necesario desarrollar un programa que los convirtiera al formato deseado.

Asimismo, y dado que es frecuente que la formulación de un problema de programación lineal posea variables fijas o elementos nulos, fue desarrollado también un preprocesamiento de la matriz del problema, que detecte éstas, así como ciertas situaciones de infactibilidad.

4.1 Formato MPS

A grandes rasgos, el formato MPS tiene las siguientes secciones, en orden de aparición:
1) NAME: el nombre del problema.
2) ROWS: cada fila o restricción tiene un nombre para distinguirla. Aquí se indica para cada una, de qué tipo de restricción se trata. Los valores posibles son G (mayor o igual), L (menor o igual), E (igual) o N (no restringida, o sea, la función objetivo).
3) COLUMNS: se listan todos los valores no nulos de la matriz del sistema, junto con los nombres de su columna y de su fila.
4) RHS: indica los valores de los términos independientes, precedidos del nombre de la restricción en la que aparece.
5) BOUNDS: sólo existe si alguna de las variables del problema está acotada. Por cada una se indica qué clase de cota es (UP: superior, LO: inferior, FX:fija), y el valor límite.
6) ENDATA: únicamente marca el fin de datos.

4.2 Conversión de formato

Para lograr la conversión de formato se desarrolló el programa LEERDATA.FOR (en Microsoft Fortran), realizando los pasos que se detallan a continuación.

En primer lugar, se recorrió el archivo de datos a convertir para contar la cantidad de filas, columnas y elementos, y calcular cuántas y cuáles eran las slacks. (Este primer paso es necesario, pues sólo conociendo estas cifras se puede reservar la memoria necesaria para los vectores de trabajo). A continuación se realizó una segunda recorrida para agregar las columnas pertenecientes a las slacks. Luego se capturaron los elementos de la matriz de restricciones, la función objetivo y los términos independientes. Por último, se transformaron las restricciones de tipo G en restricciones de tipo L, y se recorrieron las columnas ordenando los índices de fila de menor a mayor.

Finalmente, el problema expresado en el nuevo formato fue almacenado en el archivo que el operador ingresara por pantalla, listo para ser preprocesado.

Nota: En esta implementación no se consideraron problemas con cotas para las variables.

4.3 Preprocesamiento

Una vez obtenido el problema en el formato adecuado, se estudió en busca de variables o restricciones eliminables. Esto se concretó en el programa LIMPIAR.C (desarrollado en Borland C++).

En primera instancia, se recorrió la matriz del problema eliminando elementos nulos. De hallar alguno, se recalculó la estructura (los vectores IA, JA y A).

A continuación, se recorrieron todas las filas de A, buscando las siguientes situaciones:
a) La fila tiene un único elemento. En este caso, si éste y el término independiente tienen distinto signo, la variable correspondiente debería tomar un valor negativo, por lo que el problema es infactible. Si por el contrario, son de igual signo, la variable queda fijada, y la fila se elimina del problema.
b) La fila tiene múltiplos elementos, todos del mismo signo. Si el término independiente es del signo contrario, el problema es infactible. En cambio, si es nulo, todas las variables de la fila son nulas (en este caso, también se elimina la fila).
Ambos pasos se repitieron hasta que no se detectaron más cambios. Luego se recorrió la matriz en busca de filas cuyos elementos fueran proporcionales, pero cuyos términos independientes no estuvieran relacionados de igual modo. Si se hallara algun par así, el problema es infactible.

Finalmente, se grabó la estructura resultante en el archivo de salida.

4.4 Problemas de prueba

A continuación se exhiben los problemas usados en la prueba. Los datos expuestos son:
- la cantidad de filas (restricciones)
- la cantidad de columnas (variables)
- el total de columnas (incluyendo las slacks)
- la cantidad de elementos no nulos

antes y después del preprocesamiento.
La última columna indica el porcentaje de elementos de la matriz eliminados.

PROBLEMA	#FILAS		#COLUMNAS		TOTAL COLUM		#NO NULOS		%REDUC
	INI	FIN	INI	FIN	INI	FIN	INI	FIN	CION
Afiro	27	27	32	32	51	51	102	102	0.00
Adlittle	56	55	97	96	138	137	424	417	1.65
Scagr7	129	128	140	139	185	184	465	457	1.72
Blend	74	74	83	83	114	114	522	522	0.00
Sc205	205	203	203	202	317	315	665	663	0.30
Share2b	96	96	79	79	162	162	777	777	0.00
Share1b	117	112	225	220	253	248	1179	1148	2.63
Sctap1	300	300	480	480	660	660	1872	1872	0.00
Brandy	220	134	249	205	303	239	2202	1927	12.49
Scsd1	77	77	760	760	760	760	2388	2388	0.00

La resolución de estos problemas se realizó en una PC con microprocesador 30486, de una velocidad de 50 Mhz y una RAM de 4 MB. Se programó mayoritariamente en BorlandC++, bajo Windows 3.1.

4.5 Resultados de la fase uno

Para obtener un punto inicial para el algoritmo primal-dual, se corrió una versión ad-hoc, explicada en la sección 3.3, con los resultados que se presentan seguidamente. El programa desarrollado para esta fase es el FASE1.C, también desarrollado en Borland C++.

La primera columna representa el "germen" del punto inicial. El germen se obtuvo estimativamente en base a la solución óptima conocida, tomando el máximo valor dividido por la cantidad de variables.

La segunda columna muestra la cantidad de iteraciones que demandó cada problema

PROBLEMA	β	CANT.ITER
Afiro	0.50	12
Adlittle	0.10	13
Scagr7	25.0	10
Blend	0.50	09
Sc205	1.10	07
Share2b	0.40	10
Share1b	0.50	23
Sctap1	0.03	14
Brandy	0.10	21
Scsd1	0.50	02

4.6 Resultados de la fase dos

Finalmente, para la resolución propiamente dicha de cada problema, se desarrolló un conjunto de rutinas, siendo la principal FASE2.C, y las auxiliares ESTRU.C, FSIMB.C, ALCANCE.C, ACTGRADO.C, TRANSFO.C, y UNIR.C, con las cuales se realizó el ordenamiento por mínimo grado y la factorización simbólica. Todas fueron programadas en Borland C++.

Los resultados de aplicar el método primal-dual a los problemas de NETLIB mencionados, partiendo de los puntos obtenidos en la fase uno, con un parámetro α de 0.5, se exhiben en el siguiente cuadro. Las distintas columnas muestran:
- la cantidad de iteraciones y el valor de la función objetivo en la solución hallada mediante el método primal-dual,
- la cantidad de iteraciones y el valor óptimo obtenido al resolver los mismos problemas con el HyperLindo/PC de Lindo Systems

PROBLEMA	PRIMAL-DUAL		LINDO	
	ITERAC	OPTIMO	ITERAC	OPTIMO
Afiro	121	-464.7530	010	-464.7530
Adlittle	179	225495.0	134	225495.0
Scagr7	242	-2331390.0	149	-2331390.0
Blend	170	-30.8119	110	-30.81216
Sc205	268	-52.2000	142	-52.20206
Share2b	206	-415.732	084	-415.7323
Share1b	315	-76590.0	413	-76589.3
Sctap1	485	1412.25	256	1412.25
Brandy	588	1518.51	483	1518.51
Scsd1	586	8.65377	426	8.66667

En cambio, ejecutando el mismo programa para los distintos problemas, pero tomando $\alpha = 0.8$, se obtuvieron los siguientes resultados:

PROBLEMA	$\alpha = 0.5$		$\alpha = 0.8$	
	ITERAC	OPTIMO	ITERAC	OPTIMO
Afiro	121	-464.7530	117	-464.7530
Adlittle	179	225495.0	173	225495.0
Scagr7	242	-2331390.0	235	-2331390.0
Blend	170	-30.8119	165	-30.8119
Sc205	268	-52.2000	260	-52.2000
Share2b	206	-415.7320	200	-415.7320
Share1b	315	-76590.0	306	-76588.8
Sctap1	485	1412.25	472	1412.25
Brandy	588	1518.51	574	1518.51
Scsd1	586	8.65377	572	8.65377

Los resultados obtenidos fueron exactamente los mismos para 0.5 ó 0.8, excepto por la cantidad de iteraciones: el mayor parámetro insumió alrededor de un 20% menos.

La última prueba realizada fue resolver el problema AFIRO con idéntico punto de partida, pero distintos valores del parámetro. Nuevamente, a mayor parámetro, menor fue la cantidad de iteraciones necesarias para la convergencia, y por lo tanto el tiempo de resolución.

α	CANT.ITER.	SEGUNDOS
0.0001	189	32
0.001	171	28
0.01	152	25
0.1	134	22
0.3	125	21
0.5	121	21
0.8	117	19
1.0	115	19
1.5	112	18
10.0	097	16
100.0	078	13

4.7 Conclusiones

De todo lo expuesto anteriormente, se puede desprender que el método primal-dual es efectivo para resolver el problema de programación lineal, ya que obtuvo los mismos resultados que el Lindo. Sin embargo, resultó más lento que el Simplex, siendo los tiempos de resolución altamente dependientes del valor del parámetro.

En parte, puede ser que la menor velocidad de convergencia se deba a la condición de parada del primal-dual, que es independiente de la solución en cada paso. En cierto sentido, esto es una gran ventaja, puesto que es una condición universal. Sin embargo, conlleva la desventaja de no poder reducir la cantidad de iteraciones si conociéramos un punto inicial "bueno". Por ello no valía la pena ser demasiado exigentes en la fase uno (tomamos una cota de infactibilidad de 0.0005). La única forma de acelerar la convergencia es tomar mayores valores del coeficiente de actualización del parámetro α o del parámetro mismo. Esto debe hacerse con cuidado, pues un aumento en exceso perjudica a la convergencia.

Bibliografía

[AKR] I. Adler, N. Karmarkar, M. Resende & G. Veiga. *An implementation of Karmarkar's algorithm for linear programming.* Mathematical programming, Vol. 44; 1989.

[AKV] I. Adler, N. Karmarkar, M. Resende & G. Veiga. *Data structures and programming techniques for the implementation of Karmarkar's algorithm.* ORSA Journal on Computing, Vol. 1, No. 2; Spring 1989.

[Ans] K. Anstreicher. *On long step path following and SUMT for linear and quadratic programming.* Manuscript, Dept. of Operations Research, Yale University, New Haven, CT; 1989.

[Bar] E. R. Barnes. *A variation on Karmarkar's algorithm for solving linear programming problems.* Mathematical Programming, Vol. 36, pp. 174-182; 1986.

[BaL] D. Bayer and J. C. Lagarias. *The non-linear geometry of linear programming, I. Affine and projective scaling trajectories, II. Legendre transform coordinates, III. Central trajectories.* Preprints, AT&T Bell Laboratories, Murray Hill, NJ; 1986.

[Dan] G. B. Dantzig. *Maximization of a linear function of variables subject to linear inequalities.* Activity Analysis of Production and Allocation, pp. 339-347; Wiley, N. York, 1951.

[Dik] I. I. Dikin. *Iterative solution of problems of linear and quadratic programming.* Soviet Mathematics Doklady, Vol. 8, pp. 674-675; 1967.

[DoG] J. Dongarra and E. Grosse. *Distribution of mathematical software via electronic mail.* Communications of the ACM, Vol. 30, No. 5, pp. 403-407; May 1987.

[Edm] J. Edmonds. *Paths, trees and flowers.* Canadian Journal of Mathematics, Vol. 17, pp. 449-467; 1965.

[FMC] A. Fiacco & G. McCormick. *Nonlinear programming: sequential unconstrained minimization techniques.* John Wiley and Sons, New York, NY; 1968.

[Fri] K. R. Frisch. *The logarithmic potential method of convex programming.* Memorandum, University Institute of Economics, Oslo, Norway; 1955.

[Gay] D. M. Gay. *Electronic mail distribution of linear programming test problems.* Mathematical Programming Society Committee on Algorithms Newsletter 13, pp. 10-12; 1985.

[GeL] A. George & J. W. H. Liu. *Computer solution of large sparse positive definite systems.* Prentice-Hall, Englewoods Cliffs, New Jersey; 1981.

[Gon] C. C. Gonzaga. *Path following methods for linear programming.* SIAM Review, Vol. 34, No. 2, pp. 167-224; 1992.

[GoA] C. C. Gonzaga. *Algoritmos de pontos interiores em programaçao linear.* Escola Brasileira de Otimizaçao, Rio de Janeiro, Brasil; 1989.

[Kar] N. Karmarkar. *A new polynomial time algorithm for linear programming.* Combinatorica, Vol. 4, pp. 373-395; 1984.

[Kha] L. G. Khachiyan. *A polynomial algorithm for linear programming.* Soviet Mathematics Doklady, Vol. 20, pp. 191-194; 1979.

[KhT] L. G. Khachiyan & M. J. Todd. *On the complexity of approximating the maximal inscribed ellipsoid for a polytope.* Technical Report 893, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY; 1990.

[KlM] V. Klee & G. Minty. *How good is the Simplex algorithm?* Inequalities III; O. Sisha, ed.; Academic Press; New York, NY; 1972.

[Mar] A. Marxen. *Primal barrier methods for linear programming.* Technical Report, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, Ca; 1989.

[Meg] N. Meggido. *Pathways to the optimal set in linear programming.* Progress in mathematical programming - interior points and related methods, capítulo 8. Springer Verlag, Berlin; 1989.

[MHM] W. Mylander, R. Holmes, and G. McCormick. *A guide to SUMT - version 4: The computer program implementing the sequential unconstrained minimization technique for non-linear programming.* Research paper RAC-P-63, Research Analysis Corp., McLean, VA; 1971.

[Par] S.V. Parter. *The use of linear graphs in Gauss elimination.* SIAM Review 3, pp. 119-130; 1961.

[Pol] E. Polak. *Computational methods in optimization.* Academic Press, New York, NY; 1971.

[PRV] L.F. Portugal, M.G.C. Resende, G. Veiga, and J.J. Júdice. *A truncated primal-infeasible dual-feasible network interior point method*; 1994. Obtenido a través de Internet (dirección: att.com.netlib, directorio: math/people/mgcr/doc; archivo: pdnet.ps.Z).

[Ren] J. Renegar. *A polynomial-time algorithm based on Newton's method for linear programming.* Mathematical programming, 40, pp. 59-94; 1988.

[Ros] D.J. Rose. *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations.* Graph theory and computing; Academic Press, New York, NY; 1972.

[Sch] L. Schrage. *Linear, integer, and quadratic programming with LINDO.* Scientific Press, Palo Alto, CA, capítulo 22; 1984.

[Son] G. Sonnevend. *An analytical centre for polyhedrons and new classes of global algorithms for linear (smooth, convex) programming.* Lecture notes in control and information sciences 84, pp. 866-876; Springer Verlag, New York, NY; 1985.

[VMF] R. J. Vanderbei, M. J. Meketon & B. A. Freedman. *A modification of Karmarkar's linear programming algorithm.* Algorithmica, Vol. 1, pp. 395-407; 1986.