Lenguaje de Modelado Algebraico

GLPK: GNU Linear Programming Kit

OPTIMIZACIÓN DE PROBLEMAS DE PRODUCCIÓN

Depto. Investigación Operativa. Instituto de Computación. Facultad de Ingeniería, UdelaR

Índice general

Capítulo 1

Instalación

En la actualidad existe una amplia gama de paquetes informáticos que ofrecen un lenguaje de modelado algebráico, utilidades para facilitar el ingreso de datos, algoritmos eficientes (solvers), y facilidades para desplegar y analizar los resultados. Entre los más conocidos tenemos:

GLPK es un paquete informático de código abierto (open source) para la resolución de problemas de programación lineal y programación lineal entera mixta.

GAMS General Algebraic Modeling System es otro paquete informático diseñado para el modelado y resolución de problemas de programación lineal, no lineal y mixtos. Es de código propietario y su distribución es bajo el uso de licencias.

Microsoft Excel provee un componente de resolución de problemas de programación lineal y entera mixta. El componente no se instala por defecto pero puede ser fácilmente instalado a posteriori. Es de código propietario y su venta es bajo el uso de licencias.

Estas herramientas difieren principalmente en: el formato de entrada (lenguaje de programación matemático), prestaciones, algoritmos e implementación. Durante el curso utilizaremos GLPK [1] (GNU Linear Programming Kit).

En este capítulo veremos como instalar GLPK en el sistema operativo Windows XP. Los archivos de instalación pueden ser descargados de la url http://winglpk.sourceforge.net/ en formato ZIP. Debemos descomprimirlo en el directorio que deseemos (e.g. C:\Archivos de Programas) arrastrando el archivo al directorio destino (como mostramos en la figura 1.1) haciendo uso del botón derecho del mouse y eligiendo la opción *Extraer* del menú contextual.

[1] El proyecto GLPK esta disponible en la url http://www.gnu.org/software/glpk

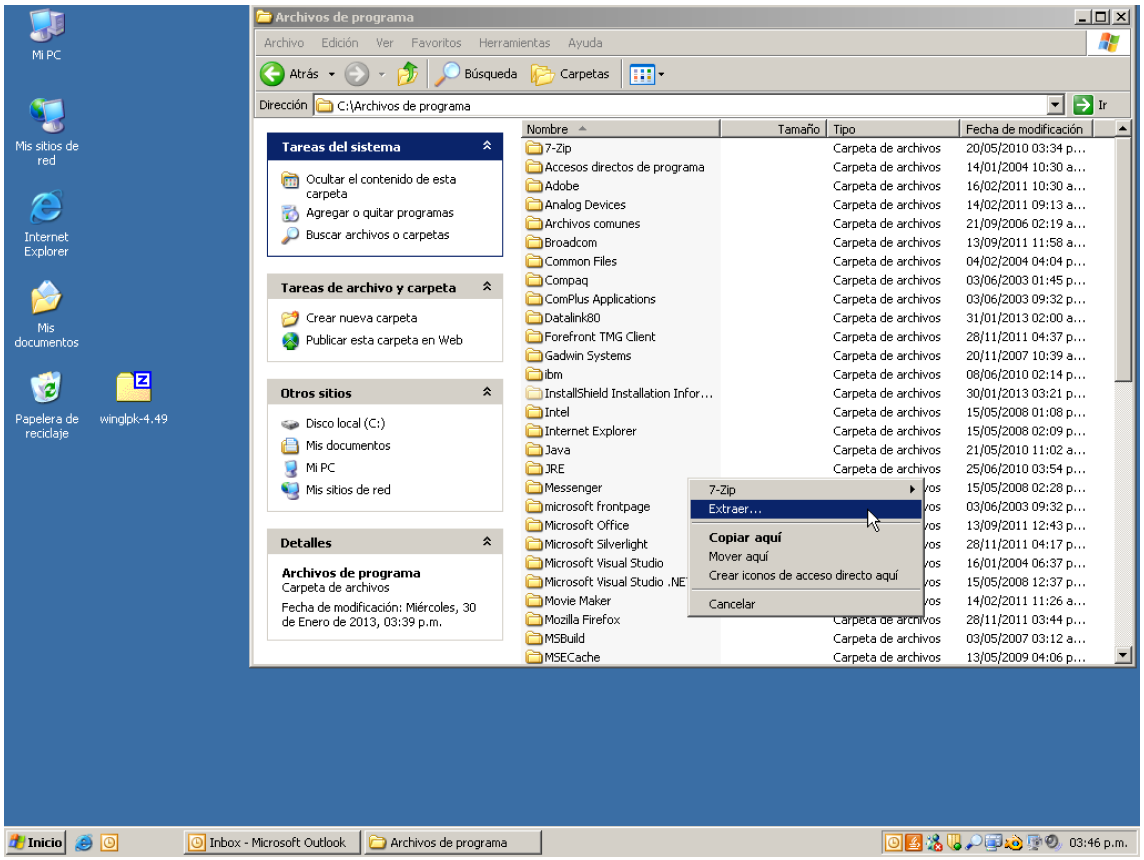

Figura 1.1: Descomprimimos el archivo winglpk-4.49.zip en el directorio C:\Archivos de Programas

Luego de descomprimido, ingresando al directorio glpk encontraremos la estructura de directorios que mostramos en la figura 1.2. Los archivos ejecutables y los enlaces a librerías dinámicas para el sistema operativo Windows de 32 bits se encuentran en el directorio w32 mientras que para la arquitectura de 64 bits se encuentran en el directorio w64. La documentación del lenguaje GMPL (GNU Math Programming Language) esta en el archivo gmpl.pdf ubicado en el directorio doc. Encontramos además varios ejemplos dentro del directorio examples.

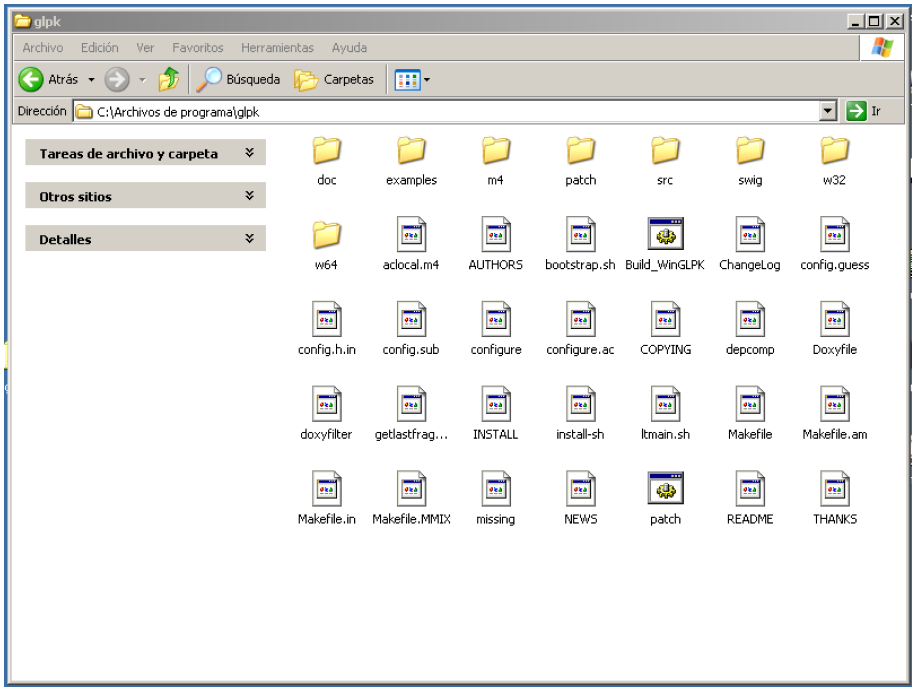

Figura 1.2: Estructura del directorio C:\Archivos de Programas\glpk

El siguiente paso es configurar la variable de entorno *Path* (que mantiene la ruta de búsqueda de programas en el sistema) para que encuentre los programas que usaremos del proyecto *GLPK*. Para eso hacemos click con el botón derecho sobre el icono de *Mi PC* y en el menú contextual que despliega hacemos click en la opción *Propiedades* como mostramos en la figura 1.3.

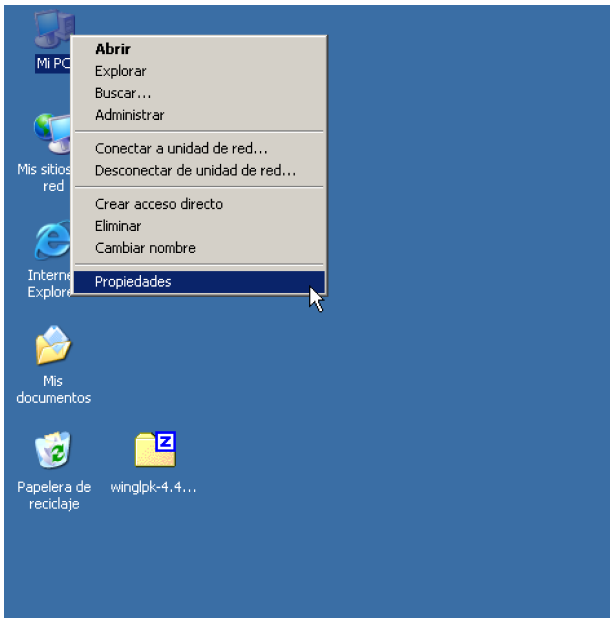

Figura 1.3: Abrir las propiedades del sistema

En la ventana que aparece, elegimos la lengüeta *Opciones Avanzadas* y luego hacemos click en el botón *Variables de entorno* (ver figura 1.4).

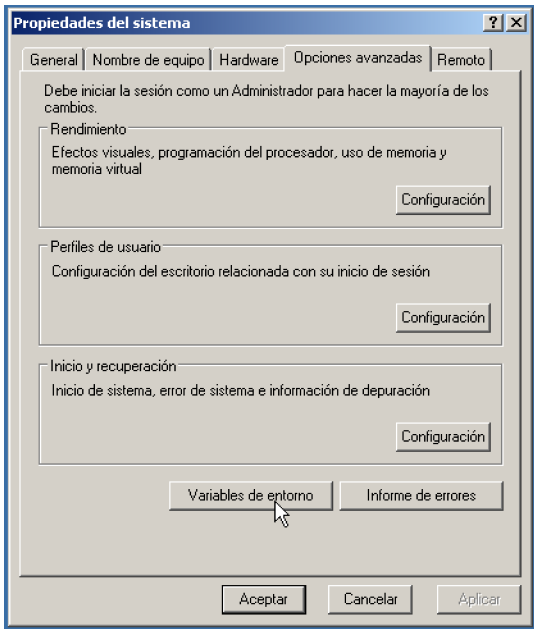

Figura 1.4: Opciones avanzadas de las propiedades del sistema

A continuación seleccionamos la variable de entorno *Path*, dentro del cuadro de las variables del sistema, y hacemos click en *Modificar* como en la figura 1.5.

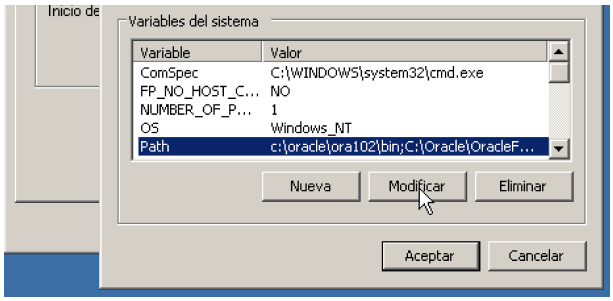

Figura 1.5: Variables del sistema

Finalmente agregamos al final del campo *Valor de variable* la ruta al directorio donde se alojan los binarios (*C:\Archivos de Programas\glpk\w32* o *C:\Archivos de Programas\glpk\w64* dependiendo de la arquitectura del hardware) separando el útimo de éste por un caracter de punto y coma (*;*). En nuestro caso estamos sobre una arquitectura de 32 bits (que es lo habitual), agregamos *;C:\Archivos de Programas\glpk\w32* como mostramos en la figura 1.6.

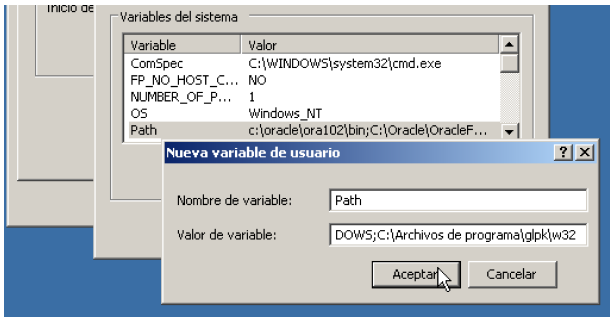

Figura 1.6: Variable de entorno Path

Finalmente elegimos *Aceptar* en las ventanas y finalizamos el proceso de instalación.

Capítulo 2

Ejecución

La ejecución del programa *glpsol.exe* se lleva a cabo desde una consola (o línea de comandos). Para tener acceso a una vamos a *Inicio*, *Ejecutar...* y allí ingresamos el comando *cmd* en el campo *Abrir* como mostramos en la figura 2.1.

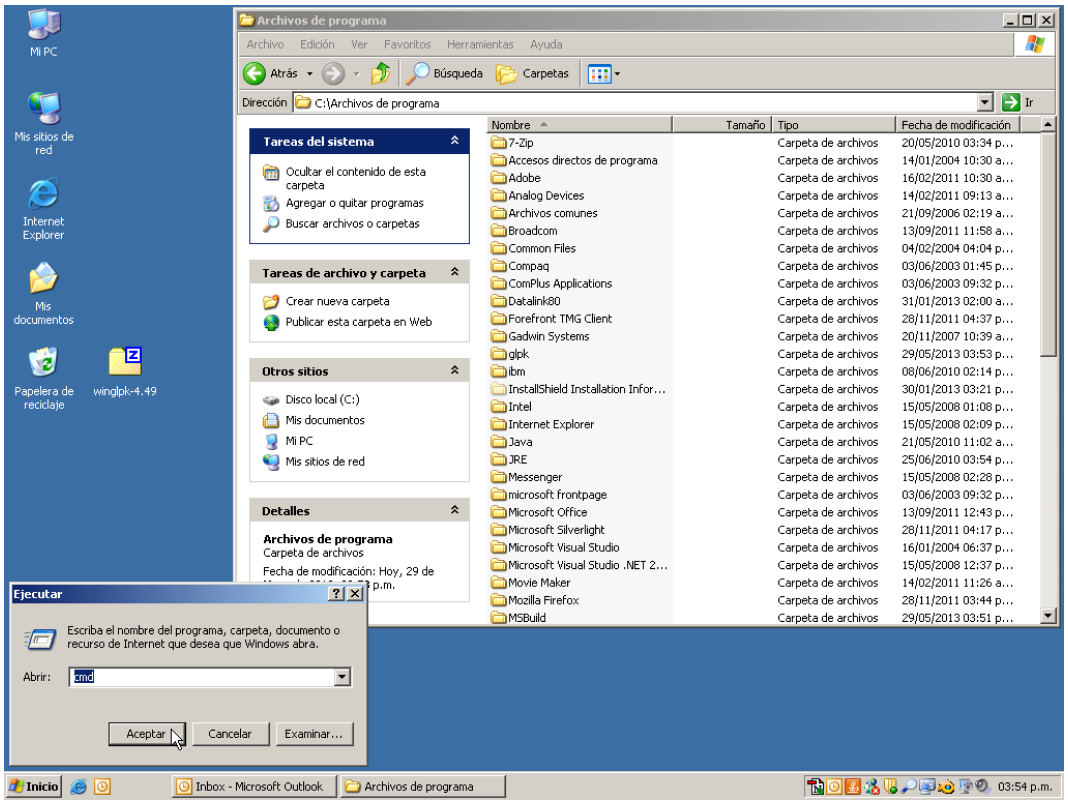

Figura 2.1: Acceso a la consola de Windows

Una vez abierta la consola ingresamos el comando que mostramos en el listado 2.1 para movernos al directorio *C:\Archivos de Programa\glpk*

Listado 2.1: Acceso al directorio glpk desde consola

```
C:\> cd "Archivos de Programa\glpk"
```

Utilizando el comando *dir* vemos el contenido del directorio *C:\Archivos de Programa\glpk* de la misma forma que lo hicimos en la figura 1.2 a través de la interfaz de usuario.

Listado 2.2: Contenido del directorio glpk desde consola

```
C:\Archivos de programa\glpk>dir
```

```
 El volumen de la unidad C no tiene etiqueta.
 El n\'mero de serie del volumen es:

 Directorio de C:\Archivos de programa\glpk

29/05/2013  03:53 p.m.    <DIR>          .
29/05/2013  03:53 p.m.    <DIR>          ..
16/04/2013  10:00 a.m.            34.967 aclocal.m4
16/04/2013  10:00 a.m.             1.629 AUTHORS
11/04/2012  11:52 p.m.                63 bootstrap.sh
16/04/2013  09:13 p.m.             4.290 Build_WinGLPK.bat
16/04/2013  10:00 a.m.            97.775 ChangeLog
16/04/2013  10:00 a.m.            44.935 config.guess
16/04/2013  10:00 a.m.               711 config.h.in
16/04/2013  10:00 a.m.            35.698 config.sub
16/04/2013  10:00 a.m.           411.309 configure
16/04/2013  10:00 a.m.             3.767 configure.ac
16/04/2013  10:00 a.m.            35.147 COPYING
16/04/2013  10:00 a.m.            23.910 depcomp
29/05/2013  03:53 p.m.    <DIR>          doc
28/10/2012  07:02 a.m.            76.086 Doxyfile
28/10/2012  07:02 a.m.                50 doxyfilter
29/05/2013  04:22 p.m.    <DIR>          examples
29/10/2009  06:47 p.m.                93 getlastfragment.awk
16/04/2013  10:00 a.m.             8.044 INSTALL
16/04/2013  10:00 a.m.            13.997 install-sh
16/04/2013  10:00 a.m.           284.668 ltmain.sh
29/05/2013  03:53 p.m.    <DIR>          m4
29/10/2009  07:03 p.m.               270 Makefile
16/04/2013  10:00 a.m.               119 Makefile.am
16/04/2013  10:00 a.m.            23.945 Makefile.in
16/04/2013  10:00 a.m.             3.951 Makefile.MMIX
16/04/2013  10:00 a.m.            10.179 missing
16/04/2013  10:00 a.m.            74.571 NEWS
29/05/2013  03:53 p.m.    <DIR>          patch
15/05/2012  02:42 a.m.               113 patch.bat
16/04/2013  10:00 a.m.             1.553 README
29/05/2013  03:53 p.m.    <DIR>          src
29/05/2013  03:53 p.m.    <DIR>          swig
16/04/2013  10:00 a.m.             5.883 THANKS
29/05/2013  03:53 p.m.    <DIR>          w32
29/05/2013  03:53 p.m.    <DIR>          w64
              27 archivos      1.197.723 bytes
              10 dirs   9.013.407.744 bytes libres
```

Finalmente utilizando el modelo que viene como ejemplo en la herramienta *transp.mod*, ejecutamos el programa *glpsol.exe* como mostramos en el listado 2.3

Listado 2.3: Ejecución del programa con el ejemplo transp.mod

```
C:\Archivos de programa\glpk>cd w32

C:\Archivos de programa\glpk\w32>glpsol.exe --model ..\examples\transp.mod --
   output ..\examples\transp.sol
GLPSOL: GLPK LP/MIP Solver, v4.49
Parameter(s) specified in the command line:
 --model ..\examples\transp.mod --output ..\examples\transp.sol
Reading model section from ..\examples\transp.mod...
Reading data section from ..\examples\transp.mod...
63 lines were read
```

```
Generating cost...
Generating supply...
Generating demand...
Model has been successfully generated
GLPK Simplex Optimizer, v4.49
6 rows, 6 columns, 18 non-zeros
Preprocessing...
5 rows, 6 columns, 12 non-zeros
Scaling...
 A: min|aij| = 1.000e+000  max|aij| = 1.000e+000  ratio = 1.000e+000
Problem data seem to be well scaled
Constructing initial basis...
Size of triangular part = 5
      0: obj =  0.000000000e+000  infeas = 9.000e+002 (0)
*     4: obj =  1.561500000e+002  infeas = 0.000e+000 (0)
*     5: obj =  1.536750000e+002  infeas = 0.000e+000 (0)
OPTIMAL SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.1 Mb (133562 bytes)
Writing basic solution to '..\examples\transp.sol'...
```

En el archivo *transp.sol* ubicado dentro del directorio *examples* encontramos la solución óptima al problema. En este caso, el mínimo es *153.675* como vemos en el listado 2.4.

Listado 2.4: Solución del problema transp.mod utilizando programación lineal

```
C:\Archivos de programa\glpk\w32>type ..\examples\transp.sol
Problem:    transp
Rows:       6
Columns:    6
Non-zeros:  18
Status:     OPTIMAL
Objective:  cost = 153.675 (MINimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 cost         B        153.675
     2 supply[Seattle]
                    NU           350                         350         < eps
     3 supply[San-Diego]
                    B            550                         600
     4 demand[New-York]
                    NL           325           325                       0.225
     5 demand[Chicago]
                    NL           300           300                       0.153
     6 demand[Topeka]
                    NL           275           275                       0.126

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 x[Seattle,New-York]
                    B             50             0
     2 x[Seattle,Chicago]
                    B            300             0
     3 x[Seattle,Topeka]
                    NL             0             0                       0.036
     4 x[San-Diego,New-York]
                    B            275             0
     5 x[San-Diego,Chicago]
                    NL             0             0                       0.009
```

```
     6 x[San-Diego,Topeka]
                       B            275             0

Karush-Kuhn-Tucker optimality conditions:

KKT.PE: max.abs.err = 0.00e+000 on row 0
        max.rel.err = 0.00e+000 on row 0
        High quality

KKT.PB: max.abs.err = 0.00e+000 on row 0
        max.rel.err = 0.00e+000 on row 0
        High quality

KKT.DE: max.abs.err = 2.78e-017 on column 1
        max.rel.err = 1.91e-017 on column 1
        High quality

KKT.DB: max.abs.err = 0.00e+000 on row 0
        max.rel.err = 0.00e+000 on row 0
        High quality

End of output
```

Capítulo 3

Formulación del Modelo

GNU MathProg es un lenguaje para modelar problemas de programación matemática. El modelo consiste de un conjunto de sentencias y bloques de datos construido por el usuario utilizando los elementos del lenguaje que veremos en las siguientes secciones.

En un proceso llamado *traducción*, el programa analiza la descripción del modelo y lo traduce de modo que puede ser utilizado para generar una instancia del problema de programación matemática o directamente para obtener una solución numérica del problema.

3.1. Problema de Programación Lineal

En MathProg el problema de programación lineal (LP) es expresado como sigue:

minimizar (o maximizar)

$$z = c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

sujeto a las restricciones (lineales)

$$\begin{aligned} L_1 \;\leq\; a_{11}x_1 + \;a_{12}x_2 + \ldots + \;a_{1n}x_n \leq U_1 \\ L_2 \;\leq\; a_{21}x_1 + \;a_{22}x_2 + \ldots + \;a_{2n}x_n \leq U_2 \\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot \\ L_m \leq a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq U_m \end{aligned}$$

y cotas

$$\begin{aligned} l_1 \leq x_1 \leq u_1 \\ l_2 \leq x_2 \leq u_2 \\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot \\ l_n \leq x_n \leq u_n \end{aligned}$$

donde x_1, x_2, ..., x_n son variables de decisión; z es la función objetivo; c_1, c_2, ..., c_n son los parámetros; a_{11}, a_{12}, ..., a_{mn} son los parámetros de las restricciones; L_1, L_2, ..., L_m son cotas inferiores; U_1, U_2, ..., U_m son cotas superiores; l_1, l_2, ..., l_n son cotas inferiores para las variables de decisión; u_1, u_2, ..., u_n son cotas superiores para las variables de decisión.

Las cotas de las variables de decisión y de las restricciones puedes ser finitas así como infinitas. Además las cotas inferiores pueden ser iguales a su correspondientes cotas superiores. Por lo tanto estan permitidos los siguientes tipos de variables de decisión y restricciones:

$-\infty < x < +\infty$	Variables no acotadas
$l \leq x < +\infty$	Variables acotadas inferiormente
$-\infty < x \leq u$	Variables acotadas superiormente
$l \leq x \leq u$	Variables acotas inferior y superiormente
$l = x = u$	Variables fijas

$-\infty < \sum a_j x_j < +\infty$	Restricciones no acotadas
$L \leq \sum a_j x_j < +\infty$	Restricciones de "mayor o igual"
$-\infty < \sum a_j x_j \leq U$	Restricciones de "menor o igual"
$L \leq \sum a_j x_j \leq U$	Restricciones acotadas inferior y superiormente
$L = \sum a_j x_j = U$	Restricciones de igualdad

Además de los problemas de programación lineal puros (LP), MathProg permite el modelado de problemas de programación lineal entera mixtra (MIP), donde algunas o todas las variables de decisión son restringidas a valores enteros o binarios.

3.2. Objetos del Modelo

La sentencia es la unidad básica de descripción del modelo. Las sentencias estan divididas en dos categorías: declarativas y funcionales. Las sentencias declarativas sirven para definir objetos del modelo mientas que las funcionales son para realizar acciones específicas.

El modelo es descrito en términos de conjuntos, parámetros, variables de decisión, restricciones y objetivos. Estos son conocidos como objetos del modelo y se incorporan al programa a través de sentencias de declaración. Cada objeto es definido a través de un nombre simbólico que lo identifica. Ese nombre simbólico nos servirá para hacer referencia al objeto en cuestión. En el Cuadro 3.1 mostramos la sintáxis utilizada para la declaración de los objetos.

Cuadro 3.1: Sintáxis en MathProg para la declaración de objetos del modelo

Definición y Sintáxis	Ejemplo
Conjunto	
set *nombre*;	**set** *alimento*;
Parámetro	
param *nombre dominio*;	**param** *precio*{*i in alimento*};
Variable de Decisión	
var *nombre dominio expresión*;	**var** *x*{*i in alimento*} >= *0*;
Restricciones	
s.t. *nombre dominio* : *expresión* = *expresión*;	
s.t. *nombre dominio* : *expresión* < *expresión*;	
s.t. *nombre dominio* : *expresión* > *expresión*;	
s.t. *nombre dominio* : *expresión* ≤ *expresión*;	
s.t. *nombre dominio* : *expresión* ≥ *expresión*;	**s.t.** *demanda*{*j in nutriente*} : *sum*{*i in alimento*} *x[i] * contenido[i,j] >= requisito[j]*;
Función Objetivo	
maximize *nombre dominio* : *expresión*;	
minimize *nombre dominio* : *expresión*;	**minimize** *costo* : *sum*{*i in alimento*} *precio[i] * x[i]*;

Los objetos del modelo, incluyendo los conjuntos, pueden ser matrices multidimensionales. Formalmente una matriz de n-dimensional *A* es el mapeo:

$$A : \Delta \rightarrow \Phi$$

, donde $\Delta \subseteq S_1 \times \ldots \times S_n$ es un subconjunto del producto Cartesiano del conjunto de índices, Φ es el conjunto de elementos de la matriz n-dimensional. En MathProg el conjunto Δ es llamado domi-

nio de subíndices (*subscript domain*). Sus elementos son n-tuplas $(i_1, \ldots, i_n)$, donde $i_1 \in S_1, \ldots, i_n \in S_n$. Si $n = 0$, el producto Cartesiano anterior tiene exactamente un elemento (es decir, 0-tupla o escalar).

Para referirnos a un objeto particular en una matriz, con $n > 0$, debemos utilizar subíndices. Por ejemplo si A es una matriz bidimensional definido sobre $I \times J$, nos referimos a un elemento a través de la notación con subíndices $A[i, j]$, donde $i \in I$ y $j \in J$. Para el caso de valores escalares (0-tupla o adimensionados) los subíndices no son necesarios.

Los comentarios son una construcción muy útil al momento de documentar nuestros modelos pues sirven para realizar anotaciones legibles. Estas anotaciones son ignoradas por el intérprete que procesa el modelo. En MathProg tenemos dos tipos de comentarios: de línea y de bloque. Los comentarios de línea comentan todo lo que se encuentra a la derecha del operador numeral (#). Los cometarios de bloques, por otro lado, comentan todo lo que se encuentran entre los operadores $/*$ y $*/$. En listado 4.1 mostramos ejemplos de su uso.

3.3. Datos del modelo

En MathProg hay dos formas de proporcionar datos a los conjuntos y parámetros del modelo. Una forma es por medio del operador de asignación **:=** en la declaración. Sin embargo en muchos casos es más práctico separar el modelo en sí mismo de los datos. Por esta razón en MathProg existen dos secciones: modelo y datos.

La sección del modelo es la parte principal de la descripción del modelo. Contiene las declaraciones de todos los objetos del modelo y es común a todos los problemas basados en ese modelo. La sección de datos es una parte opcional de la descripción del modelo y contiene datos específicos para un problema en particular. En MathProg las secciones del modelo y de datos pueden estar contenidas en un mismo archivo o en dos archivos separados. Si ambas secciones se ubican en un archivo, el archivo tendrá la estructura que se muestra a continuación:

```
sentencia;
sentencia;
    . . .
sentencia;
data;
bloque de datos;
bloque de datos;
    . . .
bloque de datos;
end;
```

Si las secciones del modelo y de datos se ubican en dos archivos separados, los mismos seguirán la siguiente estructura:

```
sentencia;
sentencia;
   . . .
sentencia;
end;
```

Archivo del modelo

```
data;
bloque de datos;
bloque de datos;
    . . .
bloque de datos;
end;
```

Archivo de datos

Nota: Si la sección de datos se ubica en un archivo separado, la palabra reservada **data** es opcional y puede ser omitida así como como el punto y coma (;) que le sigue.

Como vemos en ambos archivos, la sección de datos es una secuencia de bloques de datos. El orden que siguen los bloques de datos es arbitrario y no necesariamente el mismo con su correspondiente en la sección del modelo. A diferencia de la sección del modelo, en la sección de datos las expresiones no están permitidas.

En los cuadros 3.2 y 3.3 mostramos la sintáxis utilizada para la definición de bloque de datos de conjuntos y de parámetros respectivamente.

Cuadro 3.2: Sintáxis en MathProg para bloques de datos de conjuntos

Bloque de datos de conjuntos	Ejemplo
Asignación (:=)	
set *nombre* := $t_1, \ldots, t_n$;	**set** *alimento* := *Leche Carne Arroz Papa Tomate*;
Nota: Tanto operador de asignación (:=) como las comas (,) son opcionales. Comúnmente se usan para facilitar la lectura.	

Cuadro 3.3: Sintáxis en MathProg para bloques de datos de parámetros

Bloque de datos de parámetros	Ejemplo
Datos simples	
param *nombre* := v;	**param** *T* := *4*;
Nota: El operador de asignación (:=) es opcional. Comúnmente se usa para facilitar la lectura.	
Vectores	
param *nombre* := t_1, v_1, t_2, v_2, ... t_{n-1}, v_{n-1}, t_n, v_n;	**param** *precio* := Leche 24, Carne 190, Arroz 32, Papa 25, Tomate 50;
Matrices	
param *nombre* : c_1 c_2 ... c_n := r_1 a_{11} a_{12} ... a_{1n} r_2 a_{21} a_{22} ... a_{2n} . . . . . r_m a_{m1} a_{m2} ... a_{mn};	**param** *contenido* : Gr Pr Ca := $Leche$ 20 34 49, $Carne$ 250 180 0, $Arroz$ 2,5 67 780, $Papa$ 1 21 170, $Tomate$ 2 11 47;
Nota: El operador de asignación (:=) en la definición de matrices es obligatorio.	

Capítulo 4

Problemas de Programación Lineal

4.1. Problema de la Dieta

El problema trata de la selección de alimentos que satisfacen requisitos nutricionales a costo mínimo. Dado un conjunto de alimentos, con su información nutricional y precio, el objetivo es seleccionar las cantidades de alimentos a adquirir de forma de minimizar el costo de la dieta, mientras se cumple con requisitos nutricionales. Estos requisitos se establecen como cotas mínimas de algunos componentes nutricionales.

En el cuadro 4.1 mostramos los contenidos de nutrientes y precios de cinco alimentos, con requisitos mínimos diarios de los nutrientes:

Cuadro 4.1: Contenido de nutrientes y precios por alimento

Alimento	Grasas (g)	Proteínas (g)	Carbohidratos (g)	Precio ($)
Leche (L)	20	34	49	24
Carne (Kg)	250	180	0	190
Arroz (Kg)	2.5	67	780	32
Papa (Kg)	1	21	170	25
Tomate (Kg)	2	11	47	50
Requisitos (/día)	60	120	280	

El objetivo es determinar la cantidad diaria de cada alimento a adquirir mediante las variables x_L, x_C, x_A, x_P, x_T.

Para cada nutriente existe una restricción de cantidad mínima.

$$\begin{cases} \text{mín} & 24x_L + 190x_C + 32x_A + 25x_P + 50x_T & \\ s.a. & 20x_L + 250x_C + 2{,}5x_A + x_P + 2x_T \geq 60, & (\text{Grasas}) \\ & 34x_L + 180x_C + 67x_A + 21x_P + 11x_T \geq 120, & (\text{Proteínas}) \\ & 49x_L + 780x_A + 170x_P + 47x_T \geq 280, & (\text{Carbohidratos}) \\ & x_L, x_C, x_A, x_P, x_T \geq 0. & \end{cases} \tag{4.1}$$

4.1.1. Implementación del Modelo

En el listado 4.1 mostramos el problema de la dieta escrito en el lenguaje GMPL. En las líneas 4 y 7 definimos los conjuntos de alimentos y nutrientes respectivamente.

El siguiente paso es definir vectores para alojar los precios de los alimentos y los requisitos nutricionales diarios. La definición de los vectores las encontramos en las líneas 10 y 13 respectivamente.

Otra estructura de datos que identificamos necesaria son los contenidos de nutrientes por alimento. Una estructura acorde es una matriz bidimensional en donde accedemos al contenido nutricional de un alimento especificando el alimento y el nutriente. La definición a encontramos en la línea 16.

Definimos la función objetivo en la lína 22 y las restricciones en la línea 25.

Los datos para los distintos objetos del modelo (conjuntos, parámetros) lo asignamos en la sección data, a partir de la línea 28.

Listado 4.1: Problema de la dieta

```
# PROBLEMA DE LA DIETA
#

set alimento;
/* Alimentos */

set nutriente;
/* Nutrientes */

param precio{i in alimento};
/* Precio por alimento */

param requisito{j in nutriente};
/* Requisitos nutricionales diarios */

param contenido {i in alimento, j in nutriente};
/* Contenido nutricional j por alimento i */

var x{i in alimento} >= 0;
/* Cantidades a ser adquiridas */

minimize cost: sum{i in alimento} precio[i] * x[i];
/* Costo minimo */

s.t. demand{j in nutriente}: sum{i in alimento} x[i] * contenido[i,j] >=
   requisito[j];
/* Satisfaga los requisitos diarios para el nutriente i */

data;

set alimento := Leche Carne Arroz Papa Tomate;

set nutriente := Grasas Proteinas Carbohidratos;

param precio := Leche   24      /*($ por litro)*/
                Carne   190     /*($ por Kg)*/
                Arroz   32      /*($ por Kg)*/
                Papa    25      /*($ por Kg)*/
                Tomate  50;     /*($ por Kg)*/

param requisito := Grasas       60      /*(g diarios)*/
                Proteinas       120     /*(g diarios)*/
                Carbohidratos   280;    /*(g diarios)*/

param contenido :
                        Grasas  Proteinas       Carbohidratos :=
#                       (g)     (g)             (g)
                Leche   20      34              49
                Carne   250     180             0
                Arroz   2.5     67              780
                Papa    1       21              170
                Tomate  2       11              47;

end;
```

4.1.2. Analisis de resultados

En el listado 4.2 mostramos la salida de la ejecución del programa para el problema de la dieta. El reporte muestra que glpsol ha generado correctamente el modelo y explica brevemente como el modelo ha sido manejado internamente por GLPK. Al final hay información sobre la solución y los recursos usados por GLPK para resolverlo. En esta oportunidad la solución encontrada es óptima.

Listado 4.2: Ejecución del programa para el problema de la dieta

```
C:\Documents and settings\usuario>glpsol.exe --model taller3\dieta.mod --output
   taller3\dieta.sol
glpsol --model dieta.mod --output dieta.sol
GLPSOL: GLPK LP/MIP Solver, v4.49
Parameter(s) specified in the command line:
 --model dieta.mod --output dieta.sol
Reading model section from diet.mod...
Reading data section from diet.mod...
53 lines were read
Generating cost...
Generating demand...
Model has been successfully generated
GLPK Simplex Optimizer, v4.49
4 rows, 5 columns, 19 non-zeros
Preprocessing...
3 rows, 5 columns, 14 non-zeros
Scaling...
 A: min|aij| =  1.000e+00  max|aij| =  7.800e+02  ratio =  7.800e+02
GM: min|aij| =  2.977e-01  max|aij| =  3.359e+00  ratio =  1.128e+01
EQ: min|aij| =  8.861e-02  max|aij| =  1.000e+00  ratio =  1.128e+01
Constructing initial basis...
Size of triangular part = 3
      0: obj =   0.000000000e+00  infeas =  3.349e+00 (0)
*     3: obj =   8.031872510e+01  infeas =  0.000e+00 (0)
OPTIMAL SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.1 Mb (139657 bytes)
Writing basic solution to 'dieta.sol'...
```

La información referente a los valores óptimos de las variables de decisión se encuentran en el archivo *dieta.sol* como mostramos en el listado 4.3. La solución esta dividida en cuatro secciones:

1. Información sobre el problema y el valor óptimo de función objetivo.
2. Información sobre el estado de la función objetivo y sus restricciones
3. Información sobre la solución óptima encontrada.
4. Información sobre las condiciones de optimalidad si existen.

Para este problema en particular vemos que la solución es óptima y que el costo mínimo para la selección de alimentos es $ 80.3187251.

Analizando la tabla vemos que el estado de los requisitos nutricionales en Grasas es NL (columna St). NL significa que la restricción ha alcanzado su cota inferior. Cuando una restricción alcanza su cota, la misma impide que la función objetivo alcance un mejor valor. Cuando esto ocurre, glpsol muestra el estado de la restricción como NL o NU (acotado inferior y superior respectivamente) y muestra el valor del marginal (también conocido como valor dual o precio sombra) el cual significa cuanto mejoraría la funcion objetivo si la restricción fuera relajada por una unidad (incrementar o decrementar en uno el lado derecho de la desigualdad dependiendo si es cota superior o inferior). En este caso significa que si tuvieramos como requisitos nutricionales en Grasas el valor 59 gramos (en lugar de 60), la función objetivo mejoraría (decrementaría, ya

que estamos frente a un problema de minimización) su valor en $ 0.414343. Con los requisitos nutricionales en Proteínas, ocurre lo mismo (St es NL). Un decremento en la restricción (de 120 a 119 gramos) la función objetivo decrementaría su valor en $ 0.462151. Ambas tienen sentido, comer menos implica pagar menos. Es importante hacer estos chequeos de "sentido común"sobre los valores marginales y las restricciones.

Los requisitos nutricionales en Carbohidratos, sin embargo, son no acotados (su estado St es B) por lo que una relajación en él, no cambiará el valor óptimo de la función objetivo. Su valor dual es 0.

Finalmente, en la segunda tabla, columna Activity del reporte aparecen los valores para las variables de decisión básicas (St igual B): Leche = 2.96414 L y Arroz = 0.286853 Kg, lo que significa que deberíamos adquirir esas cantidades de Leche y Arroz diariamente para minimizar los costos y cumplir con los requisitos nutricionales. Las variables X_C, X_P, X_T son no básicas por lo que su valor es 0. Para estas variables no básicas, los costos reducidos se muestran en la columna Marginal y los mismos indican cuanto mejoraría la función objetivo si la variable de decisión asociada entrara a la base. En este caso, al tratarse de un problema de minimización y estar en la base óptima, los costos reducidos son positivos; es decir, un cambio de base aumentaría los costos.

Listado 4.3: Solución del programa para el problema de la dieta

```
Problem:    dieta
Rows:       4
Columns:    5
Non-zeros:  19
Status:     OPTIMAL
Objective:  cost = 80.3187251 (MINimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 cost         B        80.3187
     2 demand[Grasas]
                    NL            60            60                0.414343
     3 demand[Proteinas]
                    NL           120           120                0.462151
     4 demand[Carbohidratos]
                    B        368.988           280

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 x[Leche]     B        2.96414             0
     2 x[Carne]     NL             0             0                 3.22709
     3 x[Arroz]     B       0.286853             0
     4 x[Papa]      NL             0             0                 14.8805
     5 x[Tomate]    NL             0             0                 44.0876

Karush-Kuhn-Tucker optimality conditions:

KKT.PE: max.abs.err = 1.42e-14 on row 2
        max.rel.err = 1.17e-16 on row 2
        High quality

KKT.PB: max.abs.err = 0.00e+00 on row 0
        max.rel.err = 0.00e+00 on row 0
        High quality

KKT.DE: max.abs.err = 3.55e-15 on column 1
        max.rel.err = 1.39e-16 on column 5
        High quality

KKT.DB: max.abs.err = 0.00e+00 on row 0
        max.rel.err = 0.00e+00 on row 0
        High quality
```

```
End of output
```

4.2. Problema de Distribución

A continuación explicamos un problema típico de programación, extraído de [1] que puede ser formulado linealmente. El objetivo es encontrar un programa de envíos de menor costo que satisfaga los requerimientos del mercado y los suministros de las plantas. Supongamos que contamos con tres plantas de productos enlatados ubicados en en Portland (Marine), Seattle y San Diego. Las plantas pueden llenar 250, 500 y 750 latas de conserva por día, respectivamente. El distribuidor opera con cinco almacenes ubicados en New York, Chicago, Topeka (Kansas City), Dallas y San Francisco. Cada uno de los almacenes puede vender 300 latas diariamente. El distribuidor desea determinar el número de latas de conserva a ser entregadas de las tres plantas a los cincos almacenes de modo que cada almacen debería obtener tantas latas como pueda vender diariamente a costo de transporte mínimo. El problema se caracteriza por las quince posibles actividades de entregas desde las plantas a los almacenes. Como mostramos en la figura 4.1 hay quince niveles de actividad desconocidos (a ser determinados) los cuales son las cantidades a ser distribuidas entre las quince rutas. Las restricciones que todo programa de envío debe cumplir o satisfacer son las siguientes:

- el problema debe reflejar que cada uno de los almacenes recibira la cantidad de latas que necesita
- ninguna fabrica entregará más de lo que puede producir en forma diaria

Muchos programas de entregas pueden pueden existir que satisfagan estas restricciones, pero algunos tendrán costos de envios mayores a otros. El problema es entonces determinar el programa de entrega óptimo (que tenga menor costo).

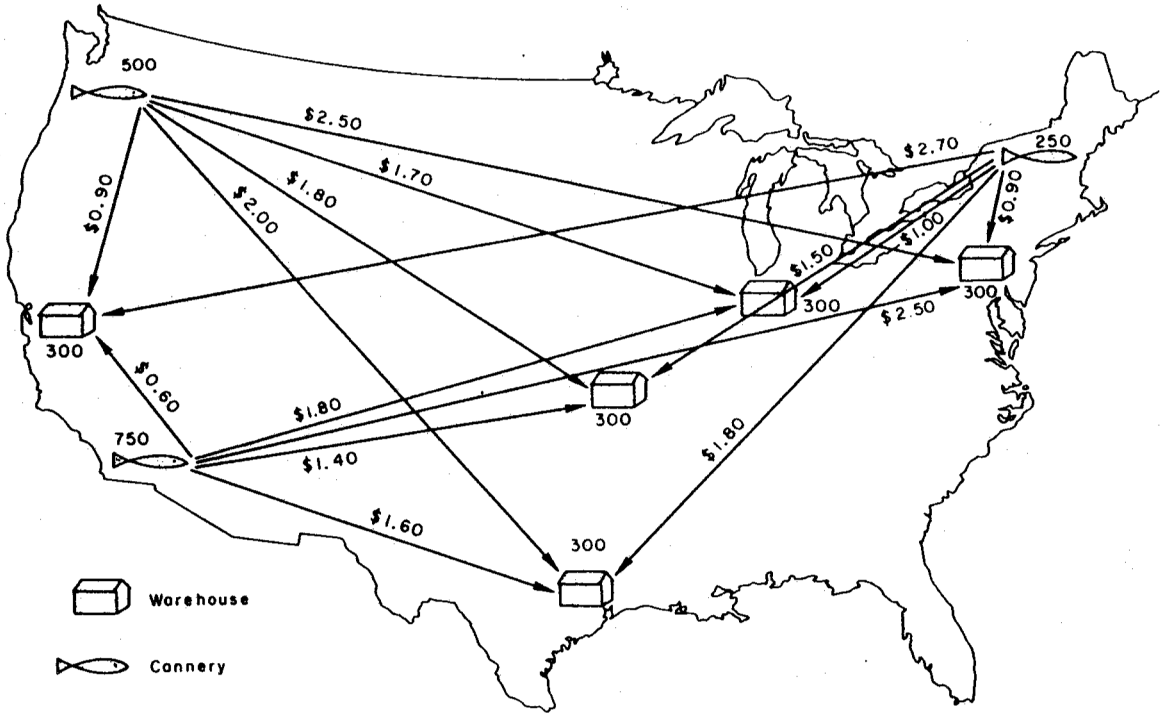

Figura 4.1: Encontrar el plan de costo mínimo de entrega desde las plantas a los almacenes (el costo por lata, oferta y demanda son las indicadas)

Para simplificar el modelo vamos a suponer que contamos con dos plantas (Seattle y San Diego) y tres almacenes (New York, Chicago y Topeka(Kansas City)). En el cuadro 4.2 mostramos los costos de envío (en dólares americanos) por unidad desde las plantas a los almacenes.

Cuadro 4.2: Costos de distribución (dólares por lata)

Plantas	Almacenes		
	New York	Chicago	Topeka
Seattle	2.5	1.7	1.8
San Diego	2.5	1.8	1.4

4.2.1. Estudio del problema

Analizando el problema encontramos las siguientes variables de decisión:

$x_{Seattle-New\,York}$	Cantidad de conservas envidadas desde Seattle a New York
$x_{Seattle-Chicago}$	Cantidad de conservas envidadas desde Seattle a Chicago
$x_{Seattle-Topeka}$	Cantidad de conservas envidadas desde Seattle a Topeka
$x_{San\ Diego-New\,York}$	Cantidad de conservas envidadas desde San Diego a New York
$x_{San\ Diego-Chicago}$	Cantidad de conservas envidadas desde San Diego a Chicago
$x_{San\ Diego-Topeka}$	Cantidad de conservas envidadas desde San Diego a Topeka

La cantidad producida en cada una de las plantas limita el plan de distribución. Para cada planta, la cantidad de conservas distribuidas debe ser menor o igual a la cantidad producida.

$$\begin{aligned} x_{Seattle-New\,York} + x_{Seattle-Chicago} + x_{Seattle-Topeka} &\leq 500 \\ x_{San\ Diego-New\,York} + x_{San\ Diego-Chicago} + x_{San\ Diego-Topeka} &\leq 750 \end{aligned}$$

Por otro lado debemos satisfacer las necesidades de los mercados por lo que agregamos las siguientes restricciones:

$$\begin{aligned} x_{Seattle-New\,York} + x_{San\ Diego-New\,York} &\geq 300 \\ x_{Seattle-Chicago} + x_{San\ Diego-Chicago} &\geq 300 \\ x_{Seattle-Topeka} + x_{San\ Diego-Topeka} &\geq 300 \end{aligned}$$

Finalmente agregamos la no negatividad de las variables de decisión:

$x_{Seattle-New\,York} \geq 0$, $x_{Seattle-Chicago} \geq 0$, $x_{Seattle-Topeka} \geq 0$,
$x_{San\ Diego-New\,York} \geq 0$, $x_{San\ Diego-Chicago} \geq 0$, $x_{San\ Diego-Topeka} \geq 0$

La función objetivo será entonces la expresada en la fórmula 4.2.

$$\text{mín} \sum_{i \in \{Seattle, San-Diego\}} \sum_{j \in \{New-York, Chicago, Topeka\}} costo\_transporte[i,j] \times x[i,j] \tag{4.2}$$

Pasando en limpio obtenemos el modelo 4.3.

$$\begin{cases} \text{mín} & \sum_{i \in \{Seattle, San-Diego\}} \sum_{j \in \{New-York, Chicago, Topeka\}} costo\_transporte[i,j] \times x[i,j] \\ s.a. & \sum_{j \in \{New\,York, Chicago, Topeka\}} x_{Seattle-j} \leq 500 \\ & \sum_{j \in \{New\,York, Chicago, Topeka\}} x_{San\ Diego-j} \leq 750 \\ & \sum_{i \in \{Seattle, San\ Diego\}} x_{i-New\,York} \geq 300 \\ & \sum_{i \in \{Seattle, San\ Diego\}} x_{i-Chicago} \geq 300 \\ & \sum_{i \in \{Seattle, San\ Diego\}} x_{i-Topeka} \geq 300 \\ & x_{i,j} \geq 0 \text{ con } i \in \{Seattle, San\ Diego\} \text{ y } j \in \{New\,York, Chicago, Topeka\} \end{cases} \tag{4.3}$$

4.2.2. Implementación del modelo

En el listado 4.4 mostramos el problema de transporte escrito en el lenguaje GMPL. En las líneas 4 y 7 definimos los conjuntos de plantas y almacenes respectivamente.

El siguiente paso es definir vectores para alojar los datos de capacidad de las plantas y demanda de los mercados. Accedemos a los elementos del primer vector utilizando como índice el conjunto de plantas mientras que para los del segundo utilizamos el conjunto de almacenes definidos anteriormente. La definición de los vectores las encontramos en las líneas 10 y 13 respectivamente.

Otra estructura de datos que identificamos necesaria es para los costos de transporte. Una estructura acorde es una matriz bidimensional en donde accedemos al costo especificando el origen (planta) y el destino (almacen). La definición la encontramos en la línea 16.

Identificados los objetos necesarios para el modelo, definimos la función objetivo en la lína 22 y las restricciones en las líneas 25 y 28.

Los datos para el modelo los asignamos en la sección data, a partir de la línea 31.

Listado 4.4: Problema de transporte

```
1  # PROBLEMA DE TRANSPORTE
2  #
3
4  set planta;
5  /* Plantas */
6
7  set almacen;
8  /* Almacenes */
9
10 param capacidad{i in planta};
11 /* Capacidad de la planta i */
12
13 param demanda{j in almacen};
14 /* Demanda del mercado j */
15
16 param transporte{i in planta, j in almacen};
17 /* Costo de transporte por lata de conserva */
18
19 var x{i in planta, j in almacen} >= 0;
20 /* Cantidades a ser distribuidas */
21
22 minimize cost: sum{i in planta, j in almacen} transporte[i,j] * x[i,j];
23 /* Costos totales de transporte */
24
25 s.t. supply{i in planta}: sum{j in almacen} x[i,j] <= capacidad[i];
26 /* Sujeto a los limites de la planta i */
27
28 s.t. demand{j in almacen}: sum{i in planta} x[i,j] >= demanda[j];
29 /* Satisfaga la demanda del mercado j */
30
31 data;
32
33 set planta := Seattle San-Diego;
34
35 set almacen := New-York Chicago Topeka;
36
37 param capacidad := Seattle     500
38                    San-Diego   750;
39
40 param demanda := New-York    300
41                  Chicago     300
42                  Topeka      300;
43
44 param transporte :              New-York   Chicago   Topeka :=
45                     Seattle     2.5        1.7       1.8
46                     San-Diego   2.5        1.8       1.4  ;
```

```
47
48 end;
```

4.2.3. Analisis de resultados

En el listado 4.5 vemos que si bien la restricción *supply[Seattle]* acota la función objetivo superiormente, el valor marginal esta muy próximo a 0 (ϵ, con $\epsilon \to 0$). Por otro lado, las restricciones de mercado nos indican que si pudieramos elegir un mercado para relajar, nos convendría elegir el de New York, dado que cada unidad decrementaría los costos 2.5 puntos en el valor óptimo, dado que es el destino más caro.

Analizando la segunda tabla vemos que el programa de envío de menor costo es aquel que envía desde Seattle 200 latas a New York, y 300 a Chicago; y desde San Diego 100 a New York y 300 a Topeka cumpliendo los límites de las plantas y las demandas de los mercados. A su vez indica que enviar latas desde Seattle a Topeka o de San Diego a Chicago aumenta el costo por lata en dólares en 0.4 y 0.1 unidades respectivamente.

Listado 4.5: Solución del programa para el problema de transporte

```
C:\Documents and settings\usuario>glpsol.exe --model taller3\transporte.mod --
    output taller3\transporte.sol
...
C:\Documents and settings\usuario>type taller3\transporte.sol

Problem:    transp
Rows:       6
Columns:    6
Non-zeros:  18
Status:     OPTIMAL
Objective:  cost = 1680 (MINimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 cost         B           1680
     2 supply[Seattle]
                    NU           500                         500         < eps
     3 supply[San-Diego]
                    B            400                         750
     4 demand[New-York]
                    NL           300           300                       2.5
     5 demand[Chicago]
                    NL           300           300                       1.7
     6 demand[Topeka]
                    NL           300           300                       1.4

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 x[Seattle,New-York]
                    B            200             0
     2 x[Seattle,Chicago]
                    B            300             0
     3 x[Seattle,Topeka]
                    NL             0             0                       0.4
     4 x[San-Diego,New-York]
                    B            100             0
     5 x[San-Diego,Chicago]
                    NL             0             0                       0.1
     6 x[San-Diego,Topeka]
                    B            300             0
```

4.2.4. Cambio en la realidad del problema

Supongamos ahora que los datos de la tabla 4.2 representan la distancia en miles de millas entre las fábricas y los almacenes. Sabemos que el precio del flete 90 dólares cada mil millas. En el listado 4.6 mostramos el programa modificado. En la línea 19 definimos el parmámetro f para almacenar el precio del flete y le asignamos el valor 90 en la línea 54 (dentro de la sección data). Los costos de transporte desde las fábricas a los almacenes en miles de dólares los obtenemos evaluando el producto entre el precio del flete y la distancia en miles de millas. Dividimos entre 1000 para obtener el costo en miles de dólares. En la línea 22 definimos una matriz de costos a través de la expresión algebraica explicada anteriormente.

En este nuevo contexto, supongamos que la capacidad de la fábrica de Seattle es de 350 latas y la de San Diego es de 600. Además la demanda de los mercados para Nueva York, Chicago y Topeka es de 325, 300 y 275 respectivamente. En la sección de datos (data) podemos modificar fácilmente estos valores sin alterar el modelo; incluso lo podríamos tener distintos datos en archivos separados y hacer referencia al mismo a través de la opción –data, como mostramos en el apéndice.

El programa mostrado en el listado 4.6 corresponde al problema inicial de transporte que vimos en el capítulo 2 (ubicado en el directorio examples/transp.mod de la instalación de la herramienta). Los resultados del mismo son los mostrados en el listado 2.4. El análisis y la interpretación de los resultados queda por cuenta del lector.

Listado 4.6: Problema de transporte modificado

```
1  # PROBLEMA DE TRANSPORTE MODIFICADO
2  #
3
4  set planta;
5  /* Plantas */
6
7  set almacen;
8  /* Almacenes */
9
10 param capacidad{i in planta};
11 /* Capacidad de la planta i */
12
13 param demanda{j in almacen};
14 /* Demanda del mercado j */
15
16 param transporte{i in planta, j in almacen};
17 /* Costo de transporte por lata de conserva */
18
19 param f;
20 /* flete en dolares por lata cada 1000 millas */
21
22 param costo{i in planta, j in almacen} := f * transporte[i,j] / 1000;
23 /* costo de transporte en miles de dolares por lata */
24
25 var x{i in planta, j in almacen} >= 0;
26 /* Cantidades a ser distribuidas */
27
28 minimize cost: sum{i in planta, j in almacen} costo[i,j] * x[i,j];
29 /* Costos totales de transporte */
30
31 s.t. supply{i in planta}: sum{j in almacen} x[i,j] <= capacidad[i];
32 /* Sujeto a los limites de la planta i */
33
34 s.t. demand{j in almacen}: sum{i in planta} x[i,j] >= demanda[j];
35 /* Satisfaga la demanda del mercado j */
36
37 data;
38
```

```
set planta := Seattle San-Diego;

set almacen := New-York Chicago Topeka;

param capacidad := Seattle     350
                   San-Diego   600;

param demanda := New-York    325
                 Chicago     300
                 Topeka      275;

param transporte :              New-York   Chicago   Topeka :=
                    Seattle     2.5        1.7       1.8
                    San-Diego   2.5        1.8       1.4  ;

param f := 90;
end;
```

4.3. Problema de Determinación de Lotes No-Capacitada

El problema de determinación de lotes no capacitada (Uncapacitated Lot-sizing Problem (ULS)) puede ser descrito como sigue. Hay un horizonte de n períodos donde, para cada período t ($t \in \{1, \ldots, n\}$), hay costos fijos de producción (f_t), costos unitarios de producción (p_t), una demanada a satisfacer ($d_t \geq 0$) y costos de depósito h_t para el almacenamiento de la producción no vendida al finalizar el peródo. A continuación identificamos los objetos del modelo

Parámetros

- f_t es el costo fijo de producir en el período t,
- p_t es el costo unitario de producción en el período t,
- h_t es el costo unitario de almacenamiento en el período t,
- d_t es la demanda en el período t.

Variables de Decisión

- x_t es la cantidad producida en el período t,
- s_t es el inventario al final del peródo t,
- $y_t = 1$ si se produce en el período t, $y_t = 0$ en otro caso.

Restricciones

- equilibrio de inventario:
  $$\begin{aligned} s_{t-1} + x_t &= d_t + s_t, \quad t \in \{1, \ldots, n\}, \\ s_0 &= 0 \qquad\qquad\qquad\qquad\quad , \end{aligned}$$
- activación de producción
  Dado que no hay una cota de producción se asume un valor grande M para la activación de los costos fijos
  $$x_t \leq My_t, \quad t \in \{1, \ldots, n\},$$

Función Objetivo

- **minimizar** $\sum_{t=1}^{n} p_t x_t + \sum_{t=1}^{n} h_t s_t + \sum_{t=1}^{n} f_t y_t$

La formulación del problema es entonces la que mostramos a continuación:

$$\begin{cases} \text{mín} & \sum_{t=1}^{n} p_t x_t + \sum_{t=1}^{n} h_t s_t + \sum_{t=1}^{n} f_t y_t \\ s.a. & s_{t-1} + x_t = d_t + s_t, \\ & x_t \leq M y_t, \\ & s_0 = 0, \\ & x_t \geq 0, \\ & s_t \geq 0, \\ & y_t \in \{0, 1\}, \\ & \forall t \in \{1, \ldots, n\}. \end{cases} \tag{4.4}$$

4.3.1. Implementación del modelo

A partir del modelo y los datos formulamos el modelo de resolución. En el listado 4.7 mostramos el modelo matemático escrito en lenguaje GMPL.

Listado 4.7: Modelo para el problema de Determinación de Lotes No-Capacitada

```
# PROBLEMA DE DETERMINACION DE LOTES NO CAPACITADA
#

set T;
/* Periodos */

param M;

param f{t in T};
/* Costo fijo de producir en el periodo t */

param p{t in T};
/* Costo unitario de produccion en el periodo t, */

param h{t in T};
/* costo unitario de almacenamiento en el periodo t */

param d{t in T};
/* demanda en el periodo t */

var x{t in T} >= 0;
/* Cantidades a producirse en el periodo t */

var s{t in (T union {0})} >= 0;
/* inventario al final del periodo t */

var y{t in T}, binary;
/* 1 si se produce en el periodo t, 0 en otro caso */

minimize costo: sum{t in T} (p[t] * x[t] + h[t] * s[t] +  f[t] * y[t]);
/* Costo minimo */

s.t. equilibro{t in T}: s[t-1] + x[t] = d[t] + s[t];
/* Equilibrio de inventario */

s.t. activ{t in T}: x[t] <= M * y[t];
/* Activacion de la produccion */

s.t. inicial: s[0] = 0;
/* Inventario inicial */

```

```
42 end;
```

Supongamos que contamos con tres realidades. Cada una consiste de 10 períodos de producción. En una primera realidad contamos con costos fijos $f_t = 750$, costos de producción unitarios de $p_t = 5$ y un costo de almacenamiento $h_t = 1$ con $t \in \{1, \ldots, 10\}$. La demanda para cada período son las del cuadro 4.3.

Cuadro 4.3: Demanda para el problema ULS

Período	d_t
1	10
2	50
3	20
4	40
5	30
6	30
7	40
8	20
9	50
10	10

En el listado 4.8 mostramos el archivo de datos asociado al modelo para la primera realidad. Si analizamos los resultados veremos que el alto costo fijo de producir en cada período f_t y los bajos costos de almacenamiento $h_t, t \in \{1, \ldots, t\}$, implica que se produzca solo en el primero y se decida por el almacenamiento de la producción en los sucesivos períodos.

Listado 4.8: Sección de datos de la realidad nº 1 para el problema de Determinación de Lotes No-Capacitada

```
1  data;
2
3  set T := 1 2 3 4 5 6 7 8 9 10;
4
5  param M := 40000;
6
7  param f default  750;
8
9  param p default 20;
10
11 param h default 1;
12
13 param d :=      1       10
14                 2       50
15                 3       20
16                 4       40
17                 5       30
18                 6       30
19                 7       40
20                 8       20
21                 9       50
22                 10      10;
23
24 end;
```

Supongamos una segunda realidad en donde los costos fijos de producción por período son inferiores $f_t = 100$ y los costos de almacenamiento $h_t = 11$. En el listado 4.9 mostramos el archivo de datos asociado a la segunda realidad. Si analizamos los resultados obtenidos veremos que los costos de almacenamiento por producto son tan altos en comparación con los costos fijos de producir en el peródo que es conveniente producir en cada período.

Listado 4.9: Sección de datos de la realidad nº 2 para el problema de Determinación de Lotes No-Capacitada

```
data;

set T := 1 2 3 4 5 6 7 8 9 10;

param M := 40000;

param f default  100;

param p default 20;

param h default 11;

param d :=      1       10
                2       50
                3       20
                4       40
                5       30
                6       30
                7       40
                8       20
                9       50
                10      10;

end;
```

Finalmente analizamos una tercera realidad en donde los costos fijos de producción para cada período es $f_t = 300$ y los costos de almacenamiento es $h_t = 5$ con $t \in \{1, \ldots, t\}$. En el listado 4.10 mostramos el archivo de datos asociado a la tercera realidad. En esta realidad, vemos que se produce para cubrir la demanda de varios meses siguientes. Esto depende de la cantidad de demanda y la relación entre los costos fijos de producción y el costo de almacenamiento.

Listado 4.10: Sección de datos de la realidad nº 3 para el problema de Determinación de Lotes No-Capacitada

```
data;

set T := 1 2 3 4 5 6 7 8 9 10;

param M := 40000;

param f default  100;

param p default 20;

param h default 11;

param d :=      1       10
                2       50
                3       20
                4       40
                5       30
                6       30
                7       40
                8       20
                9       50
                10      10;

end;
```

Apéndice: Resolución del Modelo

El programa *glpsol* es ejecutado desde la línea de comandos para resolver modelos escritos en el lenguaje de modelado MathProg. Para indicarle al programa que el archivo de entrada corresponde a una descripción de un modelo, utilizamos la opción *–model* como mostramos a continuación:

```
C:\Archivos de Programa\glpk\w32> glpsol --model foo.mod
```

Cuando contamos con una sección de datos separada del archivo de la descripción del modelo, podemos hacer referencia al mismo a través de la opción *–data*. Notar que si el modelo (foo.mod) contiene una sección de datos, esta sección es ignorada.

```
C:\Archivos de Programa\glpk\w32> glpsol --model foo.mod --data foo.dat
```

Para ver el problema que ha sido generado a partir de la descripción del modelo y los datos podemos usar la opción *–wlp* como mostramos a continuación:

```
C:\Archivos de Programa\glpk\w32> glpsol --model foo.mod --wlp foo.lp
```

en cuyo caso el problema es escrito al archivo *foo.lp* en el formato conveniente (*CPLEX LP*) para un analisis visual. En circunstancias es necesario chequear la descripción del modelo sin resolver el problema generado. En este caso podemos hacer uso de la opción *–check* como sigue:

```
C:\Archivos de Programa\glpk\w32> glpsol --check --model foo.mod --wlp foo.lp
```

Para obtener la solución numérica obtenida por el programa, hacemos uso de la opción *–output*

```
C:\Archivos de Programa\glpk\w32> glpsol --model foo.mod --output foo.sol
```

en cuyo caso la solución será escrita en el archivo *foo.sol*. Por medio del comando *type* en sistemas windows podemos ver el contenido del archivo como hicimos en el listado 2.4 y 4.5.

La lista completa de opciones de *glpsol* se encuentra en el manual de referencia incluido en la distribución GLPK (directorio: doc\glpk.pdf).

Bibliografía

[1] G.B. Dantzig. *Linear Programming and Extensiory*. Princeton landmarks in mathematics and physics : mathematics. University, 1963.