Tutorial: Resolución de Problemas de Programación Lineal en GLPK
Heurística y Optimización.
Grado de Ingeniería en Informática. Curso 2013-2014.

1. Modelo Simple

Un problema de optimización lineal consta de los siguientes aspectos:

- Variables de decisión
- Función Objetivo
- Restricciones

Analicemos el siguiente ejemplo:
Geppetto Madera SA produce dos tipos de juguetes de madera: soldados y trenes. Cada soldado se vende por 27E y usa 10E de materias primas. Cada tren se vende por 21E y usa 9E de materia prima. Además cada soldado genera un sobrecoste y unos gastos adicionales por mano de obra de 14E, mientras que los sobrecostes por tren son de 10E. La producción de juguetes de madera requiere dos procesos: carpintería y acabado. Cada soldado requiere 1 hora de carpintería y 2 horas de acabado, mientras que cada tren requiere 1 y 1. Cada semana, el taller dispone de 80 horas para tareas de carpintería y 100 horas para tareas de acabado. La demanda de trenes es ilimitada, pero como mucho se venden 40 soldados por semana. Geppetto Madera SA nos pide calcular cuántos juguetes de cada tipo producir para maximizar su beneficio.

1.1. Variables de decision

Las variables de decisión son las unidades de cada juguete, extraído de la frase "cuántos juguetes de cada tipo producir para maximizar su beneficio". La sintaxis es *var name attrib , . . . , attrib;*, por ejemplo:

```
var x;
var y binary;
var z integer, >= 0;
```

En el ejemplo se pueden representar las unidades como x1 y x2, obviando que son variables enteras por simplicidad:

```
var x1 >= 0;
var x2 >= 0;
```

1.2. Función objetivo

En el ejemplo hay que maximizar el beneficio. Éste se obtiene multiplicando las unidades producidas por el beneficio que da cada juguete.

```
maximize Profit: 3*x1 + 2*x2;
```

La única opción es usar *minimize* en vez de *maximize* cuando quiera minimizarse la función. El nombre lo elige el usuario, en este caso "Profit" (beneficio).

1.3. Restricciones

Siguen un formato similar al de la función objetivo, aunque se etiquetan con "s.t." al comienzo (opcional).

```
s.t. Finishing : 2*x1 + x2 <= 100;
s.t. Carpentry : x1 + x2 <= 80;
s.t. Demand    : x1 <= 40;
```

2. Modelo General

Mathprog permite declarar enumeraciones de objetos (sets) y características (parámetros). Esto es útil para generalizar el modelo.

2.1. Sets

Los sets son los conjuntos de objetos del problema, en este caso los juguetes:

```
set TOY;
```

Los sets pueden estar contenidos en otro set, para lo que se usa "whithin". También pueden relacionarse sets con objetos de otro set.

```
set CAR within VEHICLE;
set CITY{c in COUNTRY}, default{}; /* if not declared, the country has no cities */
```

2.2. Parámetros

Las características del problema se codifican como parámetros. Pueden estar relacionados con objetos.

```
param total_time;
param weight{v in VEHICLE};
param price{c in CAR, c in COUNTRY};
```

En el problema tendríamos los siguientes parámetros (se podrían incluir las horas totales de carpintería y acabado como parámetros no relacionados con ningún set).

```
param Finishing_hours {i in TOY};
param Carpentry_hours {i in TOY};
param Demand_toys     {i in TOY};
param Profit_toys     {i in TOY};
```

2.3. Variables de decisión, función objetivo y restricciones

Todo lo que se había incluído en el anterior modelo de forma explícita ha de referenciarse con los parámetros adecuados. Por ejemplo, en vez de tener 2 variables de decisión en total de forma explícita, ahora habrá una por juguete:

```
var units {i in TOY} >=0;
```

La función objetivo depende de las nuevas variables de decisión y del beneficio por juguete. En este caso usamos un sumatorio:

```
maximize Profit: sum{i in TOY} Profit_toys[i]*units[i];
```

Las restricciones también han de ser modificadas. Las horas disponibles se codifican de manera similar a la función objetivo, con un sumatorio y una inecualidad (100 y 80 podrían haber sido parámetros para hacer el modelo más general).

```
s.t. Fin_hours : sum{i in TOY} Finishing_hours[i]*units[i] <= 100;
s.t. Carp_hours : sum{i in TOY} Carpentry_hours[i]*units[i] <= 80;
```

La demanda no es una única restricción, sino que es una restricción por juguete (las únidades de cada juguete no deben sobrepasar la demanda de ese juguete). Esto se codifica incluyendo los sets a los que se refiere la restricción a continuación del nombre.

```
s.t. Dem {i in TOY} : units[i] <= Demand_toys[i];
```

2.4. Datos

Los datos se suelen incluir en un fichero aparte, y dan valores a los sets y parámetros del problema.

```
data;

set TOY := soldier train;

param Finishing_hours:=
soldier             2
train               1;

param Carpentry_hours:=
soldier             1
train               1;

param Demand_toys:=
soldier        40
train    6.02E+23;

param Profit_toys:=
soldier         3
train           2;

end;
```

Como no hay demanda máxima de trenes se usa un valor arbitrariamente alto (también se podría haber utilizado el atributo “default” en la definición del parámetro *Demand_toys*).

Para condensar atributos referidos a los mismos sets se pueden usar múltiples columnas:

```
param : Finishing_hours Carpentry_hours Demand_toys Profit_toys :=
soldier 2 1 40 3
train 1 1 6.02E+23 2;
```

Las matrices bidimensionales se codifican de manera similar, por ejemplo dar valores a *param price*{*c in CAR, c in COUNTRY*} sería de la siguiente manera (tened cuidado con la posición del caracter ’:’):

```
param price : Spain Italy Germany :=
Laguna       15000 15800 15200
A3            37500 38200 37000
Passat        31000 32000 37100;
```

Ejemplos de sets contenidos en otros sets (*set CAR within VEHICLE*) y de sets relacionados con otros sets (*set CITY{c in COUNTRY}*):

```
set VEHICLE := Boeing747 Laguna Chinook A3 Passat;
set CAR := Laguna A3 Passat;
set COUNTRY := Spain Italy Germany;
set CITY[Spain] := Albacete Guadalajara;
set CITY[Italy] := Napoli Brescia;
set CITY[Germany] := Hannover Heilbronn;
```

También se pueden generar expresiones más complejas para las restricciones que nos permiten representar de forma mucho más compacta una gran variedad de éstas. Por ejemplo, supongamos que tenemos una variable *ventas* que se refiere al número de unidades de cada vehículo que vendemos en cada país:

```
var ventas{i in VEHICLES, j in COUNTRY} integer, >= 0;
```

Realmente hemos declarado una matriz bidimensional de variables, y cada una de estas variables debe tomar un valor entero y mayor o igual que 0. Ahora tenemos una restricción que nos dice que las ganancias de la empresa en cada país por ventas de cada tipo de coche debe ser mayor a 100000€. Esta restricción se podría modelar de la siguiente manera:

```
s.t. benefits {i in CAR, j in COUNTRY}: ventas[i,j] * price[i,j] >= 100000;
```

que sería equivalente a modelar todas estas restricciones:

```
s.t. benefits: ventas['Laguna','Spain'] * price['Laguna','Spain'] >= 100000;
s.t. benefits: ventas['Laguna','Italy'] * price['Laguna','Italy'] >= 100000;
s.t. benefits: ventas['Laguna','Germany'] * price['Laguna','Germany'] >= 100000;
s.t. benefits: ventas['A3','Spain'] * price['A3','Spain'] >= 100000;
s.t. benefits: ventas['A3','Italy'] * price['A3','Italy'] >= 100000;
s.t. benefits: ventas['A3','Germany'] * price['A3','Germany'] >= 100000;
s.t. benefits: ventas['Passat','Spain'] * price['Passat','Spain'] >= 100000;
s.t. benefits: ventas['Passat','Italy'] * price['Passat','Italy'] >= 100000;
s.t. benefits: ventas['Passat','Germany'] * price['Passat','Germany'] >= 100000;
```

Como se puede ver, aunque la matriz *ventas* contiene variables para todos los vehículos, la anterior restricción únicamente está considerando los vehículos que se corresponden a coches, i.e., sólo aquellos vehículos que pertenecen al conjunto CAR. Ahora queremos modelar una restricción que nos diga que la suma de ventas de cualquier par de coches en un país debe ser superior a 300000€.

```
s.t. benefits_two {i in CAR, j in CAR, k in COUNTRY}: ventas[i,k] * price[i,k] + ventas[j,k] * price[j,k] >= 300000;
```

Esta restricción sería correcta, pero entre las restricciones que modela estarían las siguientes:

```
s.t. benefits_two: ventas['Laguna','Spain'] * price['Laguna','Spain'] + ventas['Laguna','Spain'] * price['Laguna','Spain'] >= 300000;
...
s.t. benefits_two: ventas['A3','Italy'] * price['A3','Italy'] + ventas['A3','Italy'] * price['A3','Italy'] >= 300000;
...
```

Es decir, la anterior restricción está sumando el beneficio de ventas de un mismo coche en un país puesto que el iterador i y el j pueden tomar el mismo valor. Si queremos considerar sólo pares de coches diferentes, entonces la anterior restricción se transformaría en:

```
s.t. benefits_two {i in CAR, j in CAR, k in COUNTRY : i <> j}: ventas[i,k] * price[i,k] + ventas[j,k] * price[j,k] >= 300000;
```

Con la expresión $i <> j$ detrás de los dos puntos obligamos a que i y j tomen valores diferentes. Detrás de estos dos puntos podemos poner expresiones lógicas más complejas del tipo $i <> j$ *and* $i < j \ldots$ Y estas expresiones lógicas también se pueden utilizar en las versiones iterativas de sumatorios, productos,...

3. Ejecución de GLPK

Con GLPK viene incluido un solver. Para ejecutarlo es necesario el siguiente comando con los ficheros correspondientes:

```
glpsol -m model.mod -d data.dat -o output.txt
```

En *output.txt* se detalla información sobre el valor final de la función objetivo, el valor que ha tomado cada una de las variables,....