UNIVERSIDAD COMPLUTENSE
MADRID

Práctica 2

Alberto García Doménech & David Gody Ruiz

Consideramos una fábrica que produce un producto final mezclando aceites refinados. Estos aceites son de dos tipos(vegetal(que tiene dos subtipos) y no vegetal(que tiene tres subtipos)). Podemos comprar aceites por un precio que varía cada mes dependiendo del tipo de aceite y vendemos el producto final por un VALOR dado. Podemos refinar MAXV de aceite vegetal al mes como máximo y MAXN de aceite no vegetal.Además el almacén tiene un límite de MCAP toneladas por aceite y cada tonelada de aceite almacenada tiene un coste de CA por mes. La dureza del producto final (suma ponderada por la cantidad de los aceites usados) tiene que estar entre MinD y MaxD. Al final del mes de junio tenemos que tener una cantidad exacta que conocemos de antemano de cada aceite en el almacén. Tenemos que conseguir un beneficio mínimo de MinB.

Hemos codificado la solución mediante 3 matrices(una para las cantidades refinadas por mes, otra para las compradas y otra para las almacenadas) por tipo de aceite(vegetal o no vegetal). Las matrices son de tamaño numeroDeMesesXnumeroTipoAceite, 6X2 en caso de aceites vegetales y 6X3 en el caso de los no vegetales. Las cantidades que aparecen en las matrices son en toneladas.

Hemos implementado las siguientes restricciones codificadas en MiniZinc:

- Hemos definido varias constraints asserts asegurándonos que los valores de entrada eran válidos

```
constraint assert ( VALOR > 0,"El valor del producto final tiene que ser
positivo");
constraint assert ( MAXV > 0,"El maximo de aceite vegetal refinado tiene
que ser positivo");
constraint assert ( MAXN > 0,"El maximo de aceite no vegetal refinado
tiene que ser positivo");
constraint assert ( MCAP > 0, "La capacidad de almacenamiento maxima tiene
que ser mayor que 0");
constraint assert(MinD < MaxD, "La dureza minima tiene que ser menor que
la dureza maxima");
constraint assert((forall (i in AceitesVeg) (cantidadVeg[i] <= MCAP)) /\
(forall (i in AceitesNoVeg) (cantidadNoVeg[i] <= MCAP)), "La cantidad
final de cada aceite no puede superar el limite de almacenamiento");
```

- Cada mes no se puede refinar más de una cantidad por aceite:

```
%refinar maximo MAXV de aceites vegetales
constraint forall (mes in meses) ((sum (aceite in AceitesVeg)
(refinVeg[mes,aceite]))<MAXV);
%refinar maximo MAXN de aceites vegetales
constraint forall (mes in meses) ((sum (aceite in AceitesNoVeg)
(refinNoVeg[mes,aceite]))<MAXN);
```

Además hemos puesto otra restricción al refinamiento y es que no se puede refinar por mes más cantidad de la que teníamos acumulada de ese aceite al principio de mes más la cantidad que hemos comprado ese mes

```
constraint forall (i in meses) (forall (aceite in AceitesVeg)
(almacVeg[i,aceite] + compVeg[i,aceite] >= refinVeg[i,aceite]));
constraint forall (i in meses) (forall (aceite in AceitesNoVeg)
(almacNoVeg[i,aceite] + compNoVeg[i,aceite] >= refinNoVeg[i,aceite]));
```

- Restricciones del almacenamiento:
  a. Al principio de mes tenemos almacenado la cantidad que tuviéramos al final del mes pasado(calculado con la suma de la cantidad inicial del mes pasado más la suma de las compras realizadas menos la resta de las cantidades refinadas)

```
%cada mes el almacenamiento equivale a la cantidad del mes anterior más
las compras hechas el mes anterior menos la cantidad refinada ese mes.
constraint forall (mes in FEBRERO..JUNIO) (forall (aceite in AceitesVeg)
(almacVeg[mes,aceite] = almacVeg[mes-1,aceite] + compVeg[mes-1,aceite] -
refinVeg[mes-1,aceite]));
constraint forall (mes in FEBRERO..JUNIO) (forall (aceite in AceitesNoVeg)
(almacNoVeg[mes,aceite] = almacNoVeg[mes-1,aceite] +
compNoVeg[mes-1,aceite]-refinNoVeg[mes-1,aceite]));
```

  b. Al final del mes de junio tenemos que tener una cantidad en concreto

```
constraint forall (aceite in AceitesVeg) (almacVeg[JUNIO,aceite]
+compVeg[JUNIO,aceite]-refinVeg[JUNIO,aceite] = cantidadVeg[aceite]);
constraint forall (aceite in AceitesNoVeg) (almacNoVeg[JUNIO,aceite]
+compNoVeg[JUNIO,aceite]-refinNoVeg[JUNIO,aceite] =
cantidadNoVeg[aceite]);
```

  c. En ningún momento podemos superar el límite de almacenamiento establecido

```
constraint forall (mes in meses, aceite in AceitesVeg)
(almacVeg[mes,aceite] <= MCAP);
constraint forall (mes in meses, aceite in AceitesNoVeg)
(almacNoVeg[mes,aceite] <= MCAP);
```

- La dureza final del producto tiene que estar entre MinD y MaxD

```
constraint forall (mes in meses) (sumaPonderada(mes)>MinD*refinMes(mes));
constraint forall (mes in meses) (sumaPonderada(mes)<MaxD*refinMes(mes));
```

Para calcular la dureza final hemos implementado dos funciones: una función sumaPonderada que dado un mes te devuelve la suma de los aceites multiplicando la cantidad que ha sido refinada ese mes por su dureza, y refinMes que calcula el total de

aceite refinado ese mes. Para evitar la división y acelerar el proceso pasamos multiplicando tanto MinD y MaxD para comparar con la sumaPonderada del mes

```
function var float:sumaPonderada(int:mes) = sum (veg in AceitesVeg)
(refinVeg[mes,veg] * durezaVeg[veg]) + sum (noveg in AceitesNoVeg)
(refinNoVeg[mes,noveg] * durezaNoVeg[noveg]);
%funcion que calcula el total de producto vendido en un mes
function var int: refinMes(int:mes) = sum (aceite in AceitesVeg)
(refinVeg[mes,aceite]) + sum (aceiteN in AceitesNoVeg)
(refinNoVeg[mes,aceiteN]);
```

- Tenemos que conseguir un beneficio superior a MinB

```
constraint benef()>MinB;
```

Para ello hemos definido una función benef que calcula el total de beneficio, multiplicando el total de aceite refinado por el valor del producto final y restando los diferentes gastos de la fábrica(almacenamiento y compras). El almacenamiento se calcula multiplicando la cantidad almacenada por el coste de almacenamiento(CA) y las compras se calculan multiplicando la cantidad comprada de cada aceite por su precio

```
function var int:benef()= sum (mes in meses) (refinMes(mes)*VALOR-
(gastos(mes)));

function var int:gastos(int:mes) = gastosCompras(mes) +
gastosAlmacenamiento(mes);
%funcion que calcula gastos en compras en un mes determinado
function var int:gastosCompras(int:mes)= (sum (aceite in AceitesVeg)
(compVeg[mes,aceite]*preciosVeg[mes,aceite]) + sum (aceite in
AceitesNoVeg) (compNoVeg[mes,aceite]*preciosNoVeg[mes,aceite]));
%funcion que calcula gastos en almacenamiento en un mes determinado
function var int:gastosAlmacenamiento(int:mes)=(sum(aceite in AceitesVeg)
(almacVeg[mes,aceite]) + sum(aceite in AceitesNoVeg)
(almacNoVeg[mes,aceite])) * CA;
```

Para conseguir el beneficio máximo pedimos al programa que maximice la función benef()

Hemos incluido también las restricciones propuestas en el apartado extensiones

- No hacer el producto final con más de K aceites

```
constraint forall (i in meses) ((sum(aceite in AceitesVeg)
(bool2int(refinVeg[i,aceite]>0)) + sum(aceite in AceitesNoVeg)
(bool2int(refinNoVeg[i,aceite]>0)))<=K);
```

- Si un mes refinamos un aceite tenemos que refinar como mínimo T toneladas de ese aceite

```
constraint forall (i in meses) ((forall (j in AceitesVeg) (refinVeg[i,j]!=
0 ->refinVeg[i,j] > T)) /\ (forall (k in AceitesNoVeg) (refinNoVeg[i,k]!
=0->refinNoVeg[i,k]>T)));
```

- Si utilizamos VEG1 o VEG2 para hacer el producto final tenemos que utilizar también ANV3

```
constraint forall(i in meses) ((refinVeg[i,VEG1]>0 \/ refinVeg[i,VEG1]>0)-
>refinNoVeg[i,ANV3]>0);
```

Además hemos implementado tres restricciones extra:

- El valor del producto final depende del valor de los aceites utilizados en la fabricación del producto. Para ello hemos creado dos arrays con los diferentes

valores de los aceites y hemos implementado una función benef2() que calcula el valor del producto final multiplicando la cantidad de cada aceite refinada por su valor en el mercado

```
array [1..6, AceitesVeg] of int:ValorVEG;
array [1..6, AceitesNoVeg] of int:ValorNoVEG;
%el beneficio se calcularia asi:
function var int:benef2() = sum (mes in meses) ((sum (aceite in
AceitesVeg) (refinVeg[mes,aceite]*ValorVEG[mes,aceite]))+(sum (aceite in
AceitesNoVeg) (refinNoVeg[mes,aceite]*ValorNoVEG[mes,aceite])) -
(gastos(mes)));
```

- Si un mes usamos el aceite vegetal VEG2 para realizar el producto no podemos utilizar el aceite ANV2 porque no se mezclan bien

```
constraint forall(i in meses) ((refinVeg[i,VEG2]>0->refinNoVeg[i,ANV2]=0)
\/ (refinNoVeg[i,ANV2]>0->refinVeg[i,VEG2]=0));
```

- Si un mes mes usamos más de Z toneladas de un aceite para realizar el producto el siguiente mes no podremos utilizar ese aceite

```
int:Z;
constraint forall (i in 1..5) ((forall (j in AceitesVeg) (refinVeg[i,j]>Z-
>refinVeg[i+1,j]=0)) /\ (forall (j in AceitesNoVeg) (refinNoVeg[i,j]>Z-
>refinNoVeg[i+1,j]=0)));
```

Hemos codificado todas las restricciones anteriores en SMT excepto las tres últimas restricciones y además hemos incluido varias más que en MiniZinc estaban definidas implicitamente:

- La cantidad de aceite refinada, comprada o almacenada no puede ser negativa y la cantidad máxima de aceite comprado al mes no puede superar el máximo de aceite refinado más la capacidad máxima de almacenamiento

```
# Constraints implicitas
# no podemos tener almacenamiento,compras o refin negativos
# el valor maximo de compras tiene que ser como max MAX(V o N) + MCAP
for i in range(numMeses):
    for j in range(numAceitesVeg):
        print(addassert(addge(almacVeg(i,j),"0")))
        print(addassert(addge(compVeg(i,j),"0")))
        print(addassert(addge(refinVeg(i,j),"0")))
        print(addassert(addle(compVeg(i,j),str((int(MCAP)+int(MAXV))))))

    for j in range(numAceitesNoVeg):
        print(addassert(addge(almacNoVeg(i,j),"0")))
        print(addassert(addge(compNoVeg(i,j),"0")))
        print(addassert(addge(refinNoVeg(i,j),"0")))
        print(addassert(addle(compNoVeg(i,j),str((int(MCAP)+int(MAXN))))))
```

Para codificar en SMT desde python hemos utilizado las funciones utilizadas en archivos usados en clase como “*torres-file.py*” y hemos definido una función que codifica la implicación entre dos cláusulas, otra función que implementa la resta de dos parámetros y una función que implementa una restricción con peso.

```
def addimplication(p,q):
    return "(=> " +p+ " " +q +" )"
```

```
def addresta(a1,a2):
    return "(- " +a1 + " "+ a2+ " )"
```

```
# peso por defecto:1 condiciones no fuertes <1
def addassertPeso(a,w):
    return "(assert-soft " +a+" :weight " +w+ ")"
```

Tenemos 3 conjuntos de datos, uno con los valores por defecto del enunciado de la práctica, otro con valores diferentes incluyendo los parámetros necesarios para las restricciones adicionales y un tercero con valores más amigables para el beneficio(mayor valor del producto final, mayor rango de dureza y menor coste de almacenamiento)

Con el primer conjunto de datos(sin incluir ninguna de las extensiones) Minizinc tarda ~600 msec en encontrar una solución que obtiene de beneficio 107810. Para hacer que Minizinc encuentre una solución válida hemos utilizado el resolutor COIN-BC 2.10.5/1.17.5 ya que después de dos horas ejecutando el programa con el resolutor por defecto (Gecode 6.3.0) no encontraba ninguna solución donde se cumpliera la restricción del beneficio mínimo.
Si usamos el mismo conjunto de datos y utilizamos las restricciones incluidas en el apartado extensiones (exceptuando la restricción de usar más de T toneladas de cada aceite si lo utilizamos) MiniZinc tarda ~1,5 segundos en encontrar una solución con beneficio de 101680.
Utilizando z3 para resolver el problema codificado en SMT y con el primer conjunto de datos el programa encuentra una solución con beneficio de 359112 en un tiempo de ~70 msec. Incluyendo las restricciones del apartado extensiones(incluso la del valor t) z3 encuentra la solución con beneficio 166281 en ~147 msec.

El segundo conjunto de datos es igual que el primero pero añadimos el parámetro Z para incluir la restricción adicional de si un mes refinamos una cantidad mayor que Z al mes siguiente no podemos utilizar ese aceite. Minizinc tarda ~2,7 segundos en encontrar la solución con valor de beneficio 101060

Con el tercer conjunto de datos obtenemos un beneficio de 481891 en Minizinc con un tiempo de ~510 msec mientras que con z3 obtenemos un beneficio de 337762 en ~160 msec