Basic Modeling for Discrete Optimization

Apuntes del curso de Coursera: Basic Modeling for Discrete Optimization, en español. Por: Juan Marcos Caicedo M. @ Universidad del Valle, Cali, Colombia

1.1.1 Primeros pasos (First Steps)

Los hermanos Guan Yu, Liu Bei, y Zhang Fei van a reclutar un ejército para vencer al Ejército de los Turbantes Amarillos. Tienen disponibilidad para contratar guereros de 4 distintas aldeas: Feng, Liu, Zhao y Jian. Cada guerrero de cada aldea tiene una fuerza, un costo, y se tiene un límite de cuantos soldados de cada aldea pueden tomar:

	Feng	Liu	Zhao	Jian
Fuerza	6	10	8	40
Costo	13	21	17	100
Límite	1000	400	500	150

Tenemos, entonces, para resolver este problema, algunas **restricciones**:

- El presupuesto que los hermanos poseen, que es de 10000.
- La cantidad máxima de soldados que podemos tomar de cada aldea.

Y poseemos un **objetivo**:

- Maximizar la fuerza total de nuestro ejército.

Guan Yu, sugiere contratar los mejores y más caros soldados: 100 soldados de la aldea Jian, para un total de 4000 de fuerza.

Zhang Fei, sugiere contratar el la mayor cantidad de los soldados baratos: 769 soldados de la aldea Feng, que nos da un total de 4614.

En este punto nos preguntamos: **¿Podemos dar una mejor respuesta? ¿Podemos hacerlo mejor?**. Podemos expresar esto como un modelo matemático:

Maximizar $6F + 10L + 8Z + 40J$ sujeto a:

- $13F + 21L + 17Z + 100J \leq \text{presupuesto}$
- $0 \leq F \leq 1000$
- $0 \leq L \leq 400$
- $0 \leq Z \leq 500$
- $0 \leq J \leq 150$
- $F, L, Z, J \in \mathbb{Z}, \text{presupuesto} = 10000$

Y para ello podemos escribri un modelo en **MiniZinc** que resuelve esto:

```
int: budget = 10000;
var 0..1000: F;
var 0..400: L;
var 0..500: Z;
var 0..150: J;

constraint 13*F + 21*L + 17*Z + 100*J <= budget;

solve maximize 6*F + 10*L + 8*Z + 40*J;

output ["F = \(F), L = \(L), Z = \(Z), J = \(J)"];
```

Detallando el modelo:

- En la primera línea, tenemos una **definición de parámetros**. Definimos un nuevo parámetro "budget"(presupuesto) que toma el valor de 10000. El modelo, donde vea la palabra budget, verá el valor de 10000.
- Entre las líneas 2 y 5 hay declaraciones de **variables de decisión**. Son 4 decisiones que se hacen: ¿Cuántos soldados contratamos de cada una de las 4 aldeas?. Son variables enteras, en las que escogemos un rango en los cuales toman valores. Esto es donde los 3 heroes deben decidir.
- En la línea 6 hay una **restricción**, que está forzando a que solo contratemos el numero de soldados que podemos adquirir con el presupuesto disponible.
- En la línea 7 tenemos la **función objetivo**, debemos maximizar la fuerza de nuestro ejército y así se representa de manera matemática.
- En la última línea tenemos una expresión de salida (output), y es lo que se imprime cuando encontramos una solución, e imprime el valor de cada variable de decisión.

Variables en MiniZinc: MiniZinc tiene dos tipos de variables: parámetros y variables de decisión:

- Parámetros: Son como las variables en un lenguaje de programación estándar. Deben ser asignadas a un valor (pero sólo una vez). Se declaran con un tipo: int, float o bool. Es un objeto matemático y su valor es inmutable. Por ejemplo, el presupuesto es de 10000, no puede tener después un valor de 18000. Sólo puede tomar un valor. (Puede llevar la palabra reservada par). Ejemplos:

```
int: i = 3;
par int: i = 3;
int: i; i = 3;
```

- Variables de decisión: Son como variables en matemáticas. Se declaran con la palabra reservada var. Estas son las variables que el solver trabajará con sus valores y probará. Son asignadas solo una vez con una expresión fija. Poseen un rango escrito: l..u (secuencia continua de enteros de l a u). Ejemplos (expresiones lógicamente equivalentes):

```
var int: i; constraint i >= 0; constraint i <= 4;
var 0..4: i;
var {0,1,2,3,4}: i;
```

También:

```
var int: i = x + 3;
var int: i; constraint i = x + 3;
```

Restricciones en MiniZinc: Restricciones con aritmética básica sonc readas usando los operadores relaciones aritméticos estándar:

```
=, !=, >, <, >=, <=
```

Se escriben con la palabra reservada constraint:

```
constraint x + y <= 10
```

Salidas (output) en MiniZinc: Tienen la forma:

```
output <list of strings>;
```

Y los literales string (string literals) son como en C: encerrados entre comillas dobles. Existen funciones como

```
\n y \t -> salto de linea y tab
show(v) -> el valor de v como string
\(v) -> muestra a v dentro de un literal de string
"casa"++"bote" -> concatenación de strings
```

Correr modelo en MiniZinc: Nuestro modelo army.mzn:

```
int: budget = 10000;
var 0..1000: F;
var 0..400: L;
var 0..500: Z;
var 0..150: J;

constraint 13*F + 21*L + 17*Z + 100*J <= budget;

solve maximize 6*F + 10*L + 8*Z + 40*J;

output ["F = \(F), L = \(L), Z = \(Z), J = \(J)"];
```

Se puede correr en la terminal usando:

```
$ minizinc army.mzn
```

Dará la siguiente salida:

```
F = 0, L = 392, Z = 104, J = 0
----------
==========
```

La fuerza total es 4752 y el tamaño es 496. Que es mejor que las propuestas de los otros héroes.

La línea ———- significa que halló una solución.

La línea ========== significa que halló **la mejor solución posible**.

Modelos (archivos) de MiniZinc deben terminar en .mzn

El modelo también se puede correr con el botón *Run* del IDE.

1.1.2 Segundo Modelo (Second Model)

Después de la contratación de los soldados, empezaron el entrenamiento. Sin embargo, algunos soldados huyeron pues el entrenamiento era muy duro. Los 3 hermanos desean saber cuantos soldados les quedaron sin contarlos uno a uno. Hicieron lo siguiente:

- Organizaron a los soldados en 5 líneas de igual tamaño y sobraron 2 soldados.
- Organizaron a los soldados en 7 líneas de igual tamaño y sobraron 2 soldados.
- Organizaron a los soldados en 12 líneas de igual tamaño y sobró 1 soldado.

Podríamos resolverlo usando el siguiente modelo de MiniZinc (count.mzn):

```
var 100..800: army;

constraint army mod 5 = 2;
constraint army mod 7 = 2;
constraint army mod 12 = 1;

solve satisfy;
```

En este caso no estamos tratando de optimizar algo en concreto, solo estamos tratando de encontrar una solución. Hay alguna solución a este problema?. Este es no es un **problema de optimización discreta**, es un **problema de satisfacción discreta**. También nótese que no hay output. El resultado de la ejecución en terminal del modelo es:

```
army = 457;
----------
```

La falta de una línea ========== indica que pueden haber otras soluciones. Si corremos:

```
$ minizinc --all-solutions count.mzn
```

Obtenemos:

```
army = 457;
----------
==========
```

Lo cuál indica que no hay otras soluciones. **Tienen que haber** 457 soldados restantes. Se puede configurar en el IDE que de todas las soluciones (predeterminado para problemas de optimización). Para problemas de satisfacción, no es predeterminado mostrar todas las soluciones. Como se había visto, el modelo no tiene output.

Sin embargo, MiniZinc, por defecto manda como output todas las variables declaradas que no están asignadas a una expresión.

En los problemas de satisfacción, no es necesario buscar una solución óptima.

Dentro de las **Restricciones** no sólamente hay igualdades o desigualdades lineales: también puede haber modulo, multiplicaciones, divisiones, distinto de ($\neq$), y restricciones mucho más complejas.

1.1.3 Tercer Modelo (Third Model)

Liu Bei, Guan Yu, y Zhang Fei deseaban que un hombre ostentoso les prestara caballos para la guerra. El hombre de los caballos se preguntó si los hermanos realmente eran talentosos para hacer buen uso de sus caballos. Les dió un problema y les prometió que les prestaría los caballos si resuelven dicho problema.

Les dió un mapa de la Dinastía Han, y debían colorar el mapa usando como máximo 4 colores, tal que las provincias adyacentes no compartieran el mismo color. Miremos cómo resolver este problema con un modelo de MiniZinc (color.mzn):

```
enum COLOR = {GREEN, BLUE, PINK, YELLOW};

var COLOR: Si;
var COLOR: Yan;
var COLOR: Yu;
var COLOR: Xu;
var COLOR: Qing;
var COLOR: Ji;
var COLOR: You;
var COLOR: Bing;
var COLOR: Yong;
var COLOR: Liang;
```

```
var COLOR: Yi;
var COLOR: Jing;
var COLOR: Yang;
var COLOR: Jiao;

constraint Liang != Yong;
constraint Yong != Yi;
constraint Yong != Jing;
constraint Yong != Si;
constraint Yi != Jing;
constraint Yi != Jiao;
constraint Jiao != Jing;
constraint Jiao != Yang;
constraint Jing != Yang;
constraint Jing != Yong;
constraint Jing != Si;
constraint Jing != Yu;
constraint Yang != Yu;
constraint Yang != Xu;
constraint Yu != Si;
constraint Yu != Yan;
constraint Yu != Xu;
constraint Xu != Yan;
constraint Xu != Qing;
constraint Yan != Si;
constraint Yan != Ji;
constraint Yan != Ji;
constraint Yan != Qing;
constraint Qing != Ji;
constraint Ji != You;
constraint Ji != Bing;
constraint Ji != Si;
constraint You != Bing;
constraint Bing != Si;

solve satisfy;
```

En el modelo se tiene un **tipo enumerado (enumerated type)** llamado

COLOR, que es hecho a partir de 4 colores: verde, azul, rosado y amarillo. Cada una de las provincias es declarada como una variable de este tipo enumerado, COLOR. Por otro lado, tenemos las restricciones que nos permiten decir que cada provincia vecina con la otra sean de distinto color. Podemos correr el modelo usando:

```
$ minizinc color.mzn
```

Obtenemos:

```
Si = BLUE;
Yan = GREEN;
Yu = PINK;
Xu = YELLOW;
Qing = BLUE;
Ji = PINK;
You = BLUE;
Bing = GREEN;
Yong = PINK;
Liang = GREEN;
Yi = BLUE;
Jing = GREEN;
Yang = BLUE;
Jiao = PINK;
----------
```

Que es, por supuesto, una solución a ese problema. Nóteseque no teniamos función output, entonces arrojó todas las variables de decisión con valores posibles que cumplen las restricciones impuestas.

En este ejemplo se ve una característica nueva de MiniZinc: enumareted types (tipos enumerados), son una manera de definir un conjunto finito de objetos que poseen nombre.

- Podemos entonces hacer que variables de decisión y parametros sean tipos enumerados.
- Los índices del array pueden ser tipos enumerados.
- También, conjuntos pueden ser tipose enumerados.

Los tipos enumerados son declarados con la palabra clave *enum*, de la siguiente manera: *enum* enum-name. Donde enum-name = { id1, ..., idn}. Podemos declarar variables que tomarán valores de tipos enumerados (como se vio en el ejemplo): var enum-name: var-name.

Si modificamos el modelo anterior, los colores reduciendolos solo a 3:

```
enum COLOR = {GREEN, BLUE, PINK};
```

Corriendolo obtenemos:

```
=====UNSATISFIABLE=====
```

Esto significa que no hay solución alguna al problema, con 3 colores.

Resumen: Tipos enumerados:

- Introducen un conjunto de objetos nombrados
- Util para tener «seguridad de tipos (type safety)» en los modelos
- Se verá más de tipos enumerados (enumerated types)

Existen modelos **insatisfactibles**. Debemos tener en cuenta que no todo modelo tiene solución.

También resolvimos un problema de coloreado de grafos (graph coloring problem), que es un problema clásico en la teoría de grafos, y tiene aplicaciones como registro de asignaciones, horarios, etc. Coloreado de grafos puro es manejado mejor por algoritmos especializados.

1.1.4 Modelos e Instancias (Models and Instances)

Nuestros tres héroes están interesados en conocer: Si hubiesen tenido más dinero, qué tan mejor ejército hubieran conseguido con respecto a su actual? A continuación, el modelo del Ejército que hicimos en lecciones pasadas, pero con un pequeño cambio: ya no especificamos que el presupuesto es 10000, lo dejamos sin especificar (armyd.mzn).

```
int: budget;
var 0..1000: F;
var 0..400: L;
var 0..500: Z;
var 0..150: J;

constraint 13*F + 21*L + 17*Z + 100*J <= budget;

solve maximize 6*F + 10*L + 8*Z + 40*J;

output ["F = \(F), L = \(L), Z = \(Z), J = \(J)"];
```

Cuando vayamos a correr el modelo, debemos darle ese parámetro de algún lado, puesto que no es una decisión, debe tener un valor asignado.. Entonces, podríamos correr nuestro modelo, añadiendole un archivo de datos (data file), llamado army.dzn:

```
$ minizinc armyd.mzn army.dzn
```

Y el contenido de army.dzn es:

```
budget = 20000;
```

La ejecución resulta en:

```
F = 243, L = 398, Z = 499, J = 0
----------
==========
```

Que sería el mejor ejército con el nuevo presupuesto de 20000.

También podemos, si deseamos, poner el archivo de datos directamente en la terminal de la manera:

```
$ minizinc armyd.mzn -D"budget = 20000;"
```

También se puede usar el IDE para proveer estos datos no especificados, al correr el modelo en *Run*, el IDE abrirá una ventana donde se debe ingresar los valores de parámetro faltantes.

Otra manera, es si se tiene cargado el archivo army.dzn, se puede seleccionar un archivo de datos (data file).

Los 3 hermanos observaron que tenían pocos soldados, entonces planearon reclutar más, pero para ello necesitarían un préstamo de dinero de algún hombre de negocios para poder conseguir más soldados. Consiguieron un hombre de negocios que les ayudaría con el préstamo.

El hombre de negocios accedió a prestarles el dinero, pero con la condición de que devolverían el préstamo sumado a unos intereses, durante 4 pagos realizados en el siguiente año. Cada pago se haría en cada estación (primavera, verano, otoño, invierno). Liu Bei quiere averiguar cuanto tienen que pagar exactamente en cada estación.

Cómo funcionará el préstamo:

- El prestamista otoga como préstamo una cantidad P, es un capital que es el saldo inicial.
- Cada cuarto de año, estación, cuatrimestre, se requiere dar un reembolso R.
- La tasa de interés I en cada cuarto de año.
- Al final del i-ésimo cuarto el saldo adeudado B_i está dado por:
    - El saldo anterior
    - Más el interés sobre el saldo anterior
    - Menos el reembolso

Un modelo en MiniZinc sobre esto:

```
% variables
var float: R;  % quarterly repayment
var float: P;  % principal initially borrowed
var 0.0 .. 2.0: I;  % interest rate

% intermediate variables
var float: B1; % balance after one quarter
var float: B2; % balance after two quarters
var float: B3; % balance after three quarters
var float: B4; % balance owing at end

constraint B1 = P * (1.0 + I) - R;
constraint B2 = B1 * (1.0 + I) - R;
constraint B3 = B2 * (1.0 + I) - R;
constraint B4 = B3 * (1.0 + I) - R;

solve satisfy;

output [
 "Borrowing ", show_float(0, 2, P), " at ", show_float(0,2,I*100.0),
 "% interest, and repaying ", show_float(0, 2, R),
  "\nper quarter for 1 year leaves ", show_float(0, 2, B4), " owing\n"
];
```

Tenemos nuevas características:

- Variables flotantes como R, P, I, B1, B2, B3 y B4.
- Variables como R, P desconocidas inicialmente (y sin rango)
- Un rango flotante en I: 0.0..2.0, interés entre 0 y 200

Ahora, hagamonos unas cuantas preguntas sobre este modelo:

Primera Instancia: Liu Bei quiere recibir como préstamo $10000 con tasa de interés del 4 %, reembolsando $2600. Cuánto debe al final?

Podemos crear un archivo de datos, loan1.dzn, que contenga:

```
I = 0.04;
P = 10000.0;
R = 2600.0;
```

Corriendo el modelo con el archivo de datos:

```
$ minizinc loan.mzn loan1.dzn
```

Obtenemos:

```
Borrowing 10000.00 at 4.00% interest, and repaying 2600.00
per quarter for 1 year leaves 657.78 owing
----------
```

En el IDE, como no hay parametros, no va a preguntar por un archivo de datos, y correrá para siempre. Debemos especificar el archivo de datos (data file) en la pestaña de Configuración, y luego darle *Run*. Si tenemos un archivo de proyecto, vamos al archivo de datos y seleccionamos *Run model with this data*. Podemos preguntarnos más acerca de este modelo:

Segunda Instancia: Liu Bei quiere recibir prestados $10000 con un interés del 4 %, y quiere que al final, no deba nada. Cuánto tiene que reembolsar cada cuarto?, tenemos otro archivo de datos loan2.dzn:

```
I = 0.04;
P = 10000.0;
B4 = 0.0;
```

Corriendo el modelo con el archivo de datos:

```
$ minizinc loan.mzn loan2.dzn
```

Obtenemos:

```
Borrowing 10000.00 at 4.00% interest, and repaying 2754.00
per quarter for 1 year leaves 0.00 owing
----------
```

Tercera Instancia: Liu Bei quiere recibir prestados $10000 y deber nada al final, además tiene la capacidad de reembolsar $3000 cada cuarto, Cuál es la tasa de interés?. Tenemos otro archivo de datos loan3.dzn:

```
P = 10000.0;
B4 = 0.0;
R = 3000.0;
```

Corriendo el modelo con el archivo de datos:

```
$ minizinc loan.mzn loan3.dzn
```

Obtenemos:

```
Borrowing 10000.00 at 7.71% interest, and repaying 3000.00
per quarter for 1 year leaves 0.00 owing
----------
```

Lo interesante es que algunos solvers son mejores para resolver modelos que otros, por ejemplo, el anterior es un modelo con flotantes, y para el segundo problema (instancia) Gecode tenía problemas en resolverlo. No todos los solvers son iguales, algunos modelos correrán para algunos solvers y otros no.

Usualmente, los archivos de datos (data files) definen valores para parametros, y pueden ser tipos enumerados también (enumerated types), por ejemplo, para el modelo del Coloreado del Mapa, se podría tener en color.mzn:

```
enum: COLOR;
```

Y en un archivo de datos color.dzn:

```
COLOR = {R,W,B,G,P};
```

Los archivos de datos en MiniZinc deben terminar en .dzn, y los Archivos de datos deben tener elementos de asignación: usualmente solo para parametros, PERO también posible para variables de decisión.

Todo parametro que no sea asignado dentro del modelo debe estar asignado en un archivo de datos (data file).

Se pueden añadir múltiples archivos de dato asignando distintos parametros/variables, de esta manera:

```
$ minizinc model.mzn d1.dzn d2.dzn
```

Resumen:

- Un modelo es una descripción formal de una clase de problemas de optimización (en nuestro caso).
- Una instancia es un problema de optimización particular
- Para volver un modelo en una instancia, añades los **datos**

1.1.5 Modelando Objetos (Modeling Objects)

Para motivar a sus soldados, Liu Bei, Guan Yu y Zhang Fei decidieron premiarse ellos mismos y a su ejército con un festival de comida. Para la comida de los hermanos, propusieron tener 3 de sus platos de comida favoritos, cada plato con su respendiente satisfacción y un tamaño de plato. Ellos necesitan determinar el numero de estos tres platos para maximizar su satisfacción culinario, sin exceder el espacio de la mesa disponible.

Para los soldados, hay 5 platos de comida distintos que se servirán en mesas más grandes. Así, entonces, los hermanos deben resolver un problema similar.

Hay un modelo en MiniZinc que resuelve este problema, introduciendo nuevas funcionalidades (banquet.mzn):

```
enum DISH;
int: capacity;
array[DISH] of int: satisf;
array[DISH] of int: size;

array[DISH] of var int: amt;

constraint forall(i in DISH) (amt[i] >= 0);
constraint sum(i in DISH) (size[i] * amt[i]) <= capacity;

solve maximize sum(i in DISH) (satisf[i] * amt[i]);

output ["Amount = ", show(amt), "\n"];
```

En el modelo hay un tipo enumerado DISH para los platos, luego la capacidad de la mesa. Se introduce entonces:

- Declaraciones de array: En la primera, se asocia cada plato con un valor de satisfacción, y así mismo el otro array con un tamaño. Hay otro array pero de variables de decisión, y es, asociada a cuanta cantidad de platos llevamos y podemos poner en la mesa. (Array de decisiones).
- Revisión de array (array lookups) (forall expressions), expresiones para todo: En la primera restricción decimos que la cantidad que llevamos de cada plato debe ser mayor o igual a cero.

- Expresiones de suma: La segunda restricción refiere a que al final, la cantidad multiplicada por el tamaño, esto sobre cada plato, debe ser menor o igual a la capacidad de la mesa. La función a maximizar también posee una expresión de suma (sum expression) que refiere a maximizar la satisfacción por su cantidad, esto para todos los platos.

Hemos visto a lo largo, nuevas funcionalidades:

- Expresiones de rango:
  - l .. u (l, u enteros)
  - tipos enumerados
- Arreglos de parametros y de variables
  - array[range] of variable declaration
- Consulta o revisión de array:
  - array-name[index-exp]
- Expresiones generadoras:
  - forall(i in range)(bool-expression)
    - for all i in range, bool-expression is true
  - sum(i in range)(expression)
    - sum over expression for all i in range

Podemos entonces correr el modelo proveyendo datos (banquet1.dzn):

```
DISH = {SNAKESOUP, GONGBAOFROGS, MAPOTOFU};
capacity = 18;
satisf = [29,19,8];
size = [8,5,3];
```

Corriendo:

```
$ minizinc banquet.mzn banquet1.dzn
```

Obtenemos:

```
Amount = [1, 2, 0]
----------
==========
```

Podemos correr el modelo proveyendo otros datos (banquet2.dzn):

```
DISH = {CHILIFISHHEAD, SAUSAGE, SEACUCUMBER, CHICKEN, FRIEDRICE};
capacity = 70;
satisf = [18, 16, 14, 13, 6];
size = [12, 10, 9, 8, 4];
```

Corriendo:

```
$ minizinc banquet.mzn banquet2.dzn
```

Obtenemos:

```
Amount = [0, 1, 0, 7, 1]
----------
==========
```

Aquí vemos como modelar objetos en MiniZinc:

- Creamos un tipo enumerado que nombra al objeto, en este caso, el plato de Comida, DISH.
- Luego creamos un array de parametros para cada atributo del objeto, en este caso, tamaño (size) y nivel de satisfacción (satisf).
- Luego creamos un array de variables de decisión (o array de decisión) para cada decisión de objetos, por ejemplo, la cantidad de platos que debemos tomar (amt).
- Después, construimos restricciones sobre los objetos usando comprensiones (comprehensions). En este caso forall y sum.
- Se debe notar que un modelo tiene varios conjuntos de objetos típicamente, no solo un modelo complicado.

En resumen:

- Los tipos enumerados representan conjuntos de objetos y lo que nos dan es la habilidad de construir archivos de datos (data files) que son independientes del tamaño. El tamaño puede cambiar con el archivo de datos.
- Teniamos arreglos (arrays) del objeto para representar atributos del objeto y las decisiones sobre el objeto.
- Para trabajar con aquellos arrays, teniamos que construir expresiones que involucraran un numero desconocido de objetos, y para ello usamos las expresiones forall y sum, ejemplos de expresiones generadoras, nos permiten construir expresiones sobre multiples objetos, de hecho, un numero de objetos que no conocemos hasta que vemos el archivo de datos (data file).
- El problema anterior, el problema del banquete en el jardín de duraznos (Peach Garden Banquet problem) es una versión del famoso problema de la Mochila (knapsack problem).

1.1.6 Arreglos y Comprensiones (Arrays and Comprehensions)

Después de lograr adquirir la caballería para el ejército, Liu Bei empezó a considerar en producir armas. Planeó producir 5 tipos de armas distintos, cada una con un nivel de poder (numérico). Hachas, espadas, picas, lanzas y bates, para poder incrementar la fuerza del ejército. También, tenía limitados recursos de hierro y madera, y así mismo, un número limitado de herreros y carpinteros que pueden trabajar la creación de las armas. Cada arma consumía una cantidad de hierro, madera, mano de obra del herrero y mano de obra del carpintero para poder producirse cada unidad. Con todo esto en mente, Liu Bei quería determinar cuantas armas puede producir para maximizar el poder total del ejército. **Concretamente:**

- Poder de la Hacha: 11
- Poder de la Espada: 18
- Poder de la Pica: 15
- Poder de la Lanza: 17
- Poder del Bate: 11

En cuanto a recursos se posee:

- 5000 unidades de Hierro
- 7500 unidades de Madera
- 4000 horas del Herrero
- 3000 horas del Carpintero

Y se tiene una especie de matriz de costos:

	Hacha	Espada	Pica	Lanza	Bate
Hierro	1.5	2.0	1.5	0.5	0.1
Madera	1.0	0.0	0.5	1.0	2.5
Horas Herrero	1.0	2.0	1.0	0.9	0.1
Horas Carpintero	1.0	0.0	1.0	1.5	2.5

Básicamente tenemos la matriz de consumo de recursos, que junto con los recursos disponibles nos genera una Restricción de Capacidad para cada recurso, y debemos optimizar la fuerza / poder total de las armas que podemos producir. (Fabricar armas que se permitan por el presupuesto de cada uno de los 4 diferentes recursos).

Se debe notar que este problema es muy similar a otros problemas que ya hemos visto: Tenemos nuestros productos, que son las armas. Y cada una de ellas consume una cierta cantidad de recursos. (Tenemos multiples recursos distintos, cada uno utilizado por las armas que se producen). Las restricciones y el objetivo son similares a otros problemas, básicamente, tenemos una restricción; que no podemos usar más recursos de los que tenemos, y nuestro objetivo es maximizar algo, le podemos llamar ganancia (profit), pero en este caso es la fuerza total (strength, poder total). Si miramos atrás a los anteriores problemas, podemos darnos cuenta que éste en específico cabe en ese mismo marco de problemas:

- En el problema donde se reclutaba el Ejército: **recursos** = presupuesto, **producto** = soldados
- En el problema del Banquete: **recursos** = espacio de la mesa, **producto** = platos de comida

Básicamente: los problemas del ejército, delbanquete, y de escoger la producción de armas son todos ejemplos de un mismo modelo, y podemos construir un modelo generico para todos ellos. Echemósle un ojo (prod-plan.mzn):

```
% products
enum PRODUCT;
% Profit per unit for each product
array[PRODUCT] of float: profit;

% resources
enum RESOURCE;
% Amount of each resource available
array[RESOURCE] of float: capacity;

% Units of each resource required to produce
%       1 units of product
array[PRODUCT,RESOURCE] of float: consumption;

% Variables: how much should we make of each product
array[PRODUCT] of var int: produce;

% Must produce a non-negative amount
constraint forall(p in PRODUCT) (produce[p] >= 0);

% Production can only use the available resources:
constraint forall(r in RESOURCE)
        (sum (p in PRODUCT)
            (consumption[p, r] * produce[p]) <= capacity[r]);

% Maximize profit
solve maximize sum(p in PRODUCT)
        (profit[p] * produce[p]);

output ["\(p): \(produce[p]\n"          | p in PRODUCT];
```

- Tenemos el tipo enumerado PRODUCT que representa los productos que queremos construir o queremos elegir, cuantos vamos a hacer.
- Y tenemos una ganancia o beneficio (profit) que obtenemos por cada uno de estos productos (maximizando fuerza, la ganancia es la fuerza)
- Y tenemos un número de recursos, el tipo enumerado RESOURCE, con estos recursos tenemos una capacidad dada (en el caso particular del problema tenemos hierro, madera, tiempo de carpintero y tiempo de herrero).
- Luego tenemos un arreglo de 2 dimensiones consumption, que dice basicamente, por cada producto, que tanto recurso consume fabricar una unidad de ese producto. Representa la tabla anterior. (Declaración array 2d)
- Ahora vienen las variables: un array de tamaño PRODUCT con enteros, llamado produce, aqui es donde el solver quiere decidir y probar por cada producto cuantas unidades se producen (para maximizar beneficio). Son las decisiones que debemos de hacer en todos los ejemplos.
- Luego viene una restricción de no-negatividad, que queremos producir una cantidad no-negativa de cada producto.
- Luego la última restricción significa que no podemos usar más de los recursos que tenemos disponibles. Decimos: por cada recurso, tenemos la restricción de capacidad, sumar por cada producto, el valor de consumo (cuanto cuesta fabricar) por la cantidad que se produce, toda esta suma por cada producto y esto debe ser menor que lo que se tiene de capacidad para un recurso r. Y esto se hace para todos los recursos. (Se añade una restricción a cada recurso).
- Finalmente tratamos de maximizar el beneficio, sumando todo en productos: multiplicando la ganancia por la cantidad de producción del producto en particular.

Tenemos en la segunda y última restricción de no utilizar más recursos de los disponibles, un recorrido o revisión de array (array lookup).

- Un array (arreglo) puede ser multidimensional. Se declara de la manera
  - array[index_set1, index_set2, ...] of type
- El index set (conjunto indice) de un array necesita ser
  - un rango de enteros o un tipo enumerado
  - una expresión de conjunto fijo cuyos valores forman un rango
- Los elementos de un array pueden ser cualquier cosa menos un arreglo, por ejemplo
  - array[PRODUCT, RESOURCE] of int: consume;
- La función propia de MiniZinc *length* devuelve la longitud de un array de una dimensión (1D array)
- Los arreglos de una dimensión son inicializados como si fueran una lista
  - profit = [400, 500];
  - capacity = [4000, 6, 2000, 60, 50];
- Los arreglos de dos dimensiones son inicializados con una sintaxis especifica:
  - empiezan con [|
  - | para separar filas (primera dimensión)
  - terminan con |]

```
consumption = [| 1.5, 1.0, 1.0, 1.0
               | 2.0, 0.0, 2.0, 0.0
               | 1.5, 0.5, 1.0, 1.0
               | 0.5, 1.0, 0.9, 1.5
               | 0.1, 2.5, 0.1, 2.5 |];
```

- Arreglos en cualquier dimensión (≤ 6) pueden ser inicializados usando la familia de funciones array$n$d que convierte un array de 1D a un array de nD, por ejemplo equivalentemente:

```
consumption = array2d(1..5, 1..4,
   [1.5, 1.0, 1.0, 1.0,
    2.0, 0.0, 2.0, 0.0,
    1.5, 0.5, 1.0, 1.0,
    0.5, 1.0, 0.9, 1.5,
    0.1, 2.5, 0.1, 2.5]);
```

- MiniZinc provee *array comprehensions* (arrays por comprensión), como en Haskell y ML

- Un array comprehension (array por comprensión) es de la forma

```
[ expr | generator1, generator2, ... ]
[ expr | generator1, generator2, ... where test]
```

- Ejemplo:

```
[i+j | i, j in 1..4 where i < j]
= [1+2, 1+3, 1+4, 2+3, 2+4, 3+4]
= [3, 4, 5, 5, 6, 7]
```

- Arreglos en cualquier dimensión (≤ 6) pueden ser inicializados usando la familia de funciones array$n$d que convierte un array de 1D a un array de nD, por ejemplo equivalentemente:

```
consumption = array2d(1..5, 1..4,
 [1.5, 1.0, 1.0, 1.0, 2.0, 0.0, 2.0, 0.0, 1.5, 0.5,
  1.0, 1.0, 0.5, 1.0, 0.9, 1.5, 0.1, 2.5, 0.1, 2.5]);
```

- Mientras que array$n$d convierte un array de una dimensión (1D) a uno de dimensión nD, podemos aplanar un array de dimensión nD a una lista (array de 1D) usando comprensión, por ejemplo

```
array[1..20] of int: list = [consumption[i,j] | i in 1..5, j in 1..4];
```

- Para la Iteración, es realizada mediante funciones de MiniZinc, que operan sobre una lista o sobre un conjunto:
  - Listas de números: sum, product, min, max
  - Listas de restricciones: forall, exists
- MiniZinc provee una sintaxis especial para llamados a estas funciones (y otras funciones generadoras), por ejemplo

  ```
  forall(i,j in 1..10 where i < j)
        (a[i] != a[j])
  ```

  es equivalente a (un argumento forall)

  ```
  forall([a[i] != a[j]
       | i,j in 1..10 where i < j])
  ```

Volviendo al problema original de producción de armas, aquí tenemos nuestro archivo de datos (datafile) prod-plan-weapon.dzn:

```
PRODUCT = {AXE, SWORD, PIKE, SPEAR, CLUB};
profit = [11.0, 18.0, 15.0, 17.0, 11.0];
RESOURCE = {IRON, WOOD, SMITH, CARPENTER};
capacity = [5000, 7500, 4000, 3000];
consumption = [| 1.5, 1.0, 1.0, 1.0
               | 2.0, 0.0, 2.0, 0.0
               | 1.5, 0.5, 1.0, 1.0
               | 0.5, 1.0, 0.9, 1.5
               | 0.1, 2.5, 0.1, 2.5 |];
```

Vemos un array de 2D constante, con una fila por cada producto y una columna por cada recurso

Correr el modelo con esos datos nos MiniZinc arroja:

```
AXE: 0
SWORD: 632
PIKE: 2340
SPEAR: 440
CLUB: 0
```

```
PROFIT: 53956.0
----------
==========
```

Pero recordemos que dijimos que en general este modelo funciona para otros problemas que hemos visto, por ejemplo, contratando un ejército, donde tenemos el producto que son las 4 aldeas de donde podemos contratar soldados, el beneficio por soldado, tenemos un solo recurso que es el dinero que podemos gastar, y luego el costo por soldado (consumption) que es un array 2d pero como solo tenemos un recurso entonces solo tenemos una fila por producto y solo una columna. prod-plan-army.dzn:

```
PRODUCT = {F, L, Z, J};
profit = [6.0, 10.0, 8.0, 40.0];
RESOURCE = {MONEY};
capacity = [10000.0];
consumption = [| 13.0 | 21.0 | 17.0 | 100.0 |];
```

Así mismo el problema de la comida en el jardín de duraznos (Peach Garden Banquet). prod-plan-banquet.dzn:

```
PRODUCT = {SNAKESOUP, GONGBAOFROGS, MAPOTOFU};
profit = [29.0, 19.0, 8.0];
RESOURCE = {SIZE};
capacity = [18.0];
consumption = [| 8.0 | 5.0 | 3.0 |];
```

Nota acerca de los solvers: Usar el solver Gecode (predeterminado) para el problema de producción de armamento toma un tiempo prologando, encontrando muchas soluciones antes de la mejor, sin embargo, si se usa el solver de MIP, llamado G12 MIP, es casi instantánea la solución. El problema de producción es un problema de programación entera mixta, asi que solvers MIP son ideales para este tipo de problemas.

Lo que hemos visto hasta ahora es como construir un modelo que se ajusta a datos de diferente tamaño, y es importante porque el mundo real esta lleno de resolver el mismo problema una y otra vez y los datos cambian todo el tiempo.

MiniZinc usa tipos enumerados (enumerated types) para nombrar objetos, y se usan arreglos (arrays) para capturar información acerca de estos objetos. Las comprensiones (comprehensions) ayudan a construir restricciones y expresiones acerca de datos con distinto tamaño.

1.1.7 Restricciones Globales (Global Constraints)

Para conformar el ejército, Liu Bei ha incurrido en gastos severos. Así que le preguntó a Zhang Fei que determine cuánto dinero han gastado hasta ahora. Zhang Fei ha usado varillas de conteo para sus cálculos. A punto de terminar, para su gran sorpresa, un gato saltó a través de la ventana y desorganizó las varillas de conteo. Esto dejó parte de los cálculos incompletos. Zhang Fei sabía cuantas varillas se habian involucrado en el cálculo, y no tenía idea de como recuperar su trabajo. El cálculo era importante para futura referencia, así que confundido le contó a Liu Bei el accidente.

Zhang Fei recuerda que los dígitos que quedaron desorganizados eran distintos, y tiene 12 barras que se desubicaron, junto con la parte del cálculo que aún conserva. El resultado es el siguiente:

```
   23?3
+  ?98?
=  ?3?1
```

El modelo es el siguiente. rods.mzn:

```
set of int: DIGIT = 1..9;
array[DIGIT] of int: rods = [1,2,3,4,5,2,3,4,5];

var DIGIT: M1; % first messed up digit
var DIGIT: M2; % second messed up digit
var DIGIT: M3; % third messed up digit
var DIGIT: M4; % fourth messed up digit
var DIGIT: M5; % fifth messed up digit

constraint rods[M1] + rods[M2] + rods[M3] + rods[M4] + rods[M5] = 12;

constraint 2303 + M1 * 10 + 980 + M2 * 1000 + M3 = 301 + M4 * 1000 + M5 * 10;
```

Tenemos un conjunto de dígitos del 1 al 9, y tenemos un arreglo de cuantas varas cuesta representar cada número. En el ejemplo, para representar el 1 cuesta 1 vara, el 2 cuesta 2, el 3 cuesta 3, el 4 cuesta 4, el 5 cuesta 5, y luego el 6 cuesta 2, el 7 cuesta 3, el 8 cuesta 4 y el 9 cuesta 5.

Luego tenemos 5 cosas distintas que debemos saber, cada uno de los 5 dígitos perdidos.

Después, la primera restricción es que el numero de varas usadas para los numeros desconocidos debe ser estrictamente igual a 12, que es el número de varas que fueron desacomodadas.

Luego, la última restricción es que la restricción de la suma con los digitos desconocidos se cumpla.

Vemos que en el modelo realizamos una *variable lookup*, que es cómo una revisión de una variable de decisión en un arreglo, es decir, en el caso particular, estamos viendo por ejemplo en la primera restricción cuántas varas se necesitan para representar M1, M2, hasta M5.

Luego, para garantizar que los dígitos que fueron perdidos, son todos distintos, añadimos las restricciones:

```
% alldifferent messed up digits
constraint M1 != M2;
constraint M1 != M3;
constraint M1 != M4;
constraint M1 != M5;
constraint M2 != M3;
constraint M2 != M4;
constraint M2 != M5;
constraint M3 != M4;
constraint M3 != M5;
constraint M4 != M5;
```

Sin embargo, esto resulta ser tedioso y largo, y además se puede incurrir en errores, se te puede olvidar una restricción. Hacerlo de una manera distinta, introduciendo una restricción llamada «alldifferent». Es un ejemplo de una **Restricción global**.

```
% alldifferent messed up digits
include "alldifferent.mzn";

constraint alldifferent([M1,M2,M3,M4,M5]);

solve satisfy;
```

La definición de la restricción se encuentra en otro archivo, y debemos incluirla con *include*.

Vimos unas cuantas cosas distintas en este modelo:

- Un conjunto de parametros puede ser definido como:
  - set of type: name = fixed-set;
  - pueden ser usados en lugar de un conjunto fijo
- Variable lookups (revisión de variables)
  - Podemos usar variables de decisión como parte de expresiones indexadas (index expressions)
  - Esto nos da una poderosa capacidad de modelamiento
  - En esta ocasión lo usamos de manera muy simple (basicamente por cada digito perdido, usamos el array para mirar cuantas varas toma representar ese digito)
  - Más ejemplos luego
- Declaración «include»:
  - include "file-name"
  - Incluirá el archivo a tu modelo de MiniZinc
  - Buscará en la carpeta actual y en la dirección de librería de MiniZinc
- Las declaraciones «include» son usadas para:
  - Fragmentar modelos largos en piezas pequeñas
  - Cargar definiciones de librerías de restricciones globales
  - Controlar qué definición de restricción global cargará MiniZinc para un solver en particular

Acerca de las restricciones globales:

- Técnicamente cualquie rrestricción puede tomar un numero de variables sin limites, como entrada

    - Entonces las restricciones lineales son «globales»
- Las restricciones globales son
    - Restricciones que surgen en varios problemas
- Las restricciones globales permiten
    - Hacer modelos más pequeños
    - Resolverlos de manera más facil (ya que los solvers usan la información de la estructura)

En resumen:

- Las restricciones Globales son uno de los conceptos más poderosos que se verán en el curso
    - Mucho más acerca de ellos
- Similarmente usando variables de decisión como indices en expresiones de arreglos, también es una herramienta poderosa de modelaje