Manual Python-Constraint

November 11, 2019

Un problema de satisfacción de restricciones se define como una terna (X, D, C) donde $X = \{x_i\}_{i=1}^n$ es el conjunto de variables; $D = \{D_i\}_{i=1}^n$ representa los dominios de cada variable respectivamente y $C = \{C_i\}_{i=1}^m$ es el conjunto de restricciones del problema.

La librería *python-constraint* (`https://labix.org/python-constraint`) es una librería desarrollada en *python* que permite el modelado de problemas de satisfacción de restricciones. En este manual, se explica su instalación, y la implementación de modelos de satisfacción de restricciones utilizando esta librería.

1 Instalación

Para instalar la librería, simplemente tendremos que ejecutar desde un terminal:

```
pip install python-constraint
```

Al finalizar la ejecución, por terminal debería aparecer un mensaje informando que el paquete se ha instalado correctamente. También podemos comprobar este hecho ejecutando el comando:

```
pip list
```

que lista todos los paquetes instalados, entre los cuáles deberíamos encontrar el que acabamos de instalar *python-constraint*.

2 Estructura básica de los programas con python-constraint

La estructura básica de los programas que utilizan esta librería es la siguiente:

1. Importación de la librería.
2. Definición de una variable como nuestro problema.
3. Creación de las variables junto con sus dominios.
4. Creación de las restricciones de nuestro problema.
5. Recuperación de la solución o soluciones.

Siguiendo la estructura anterior, un ejemplo sencillo de programa que emplea esta librería sería el siguiente:

```
# Importacion de la libreria
from constraint import *

# Definicion de una variable como nuestro problema
problem = Problem()

# Creacion de las variables
problem.addVariable('a', [1, 2])
problem.addVariable('b', [1, 2])

# Creacion de las restricciones
problem.addConstraint(lambda a, b: a != b, ('a', 'b'))

# Recuperacion de las soluciones
problem.getSolutions()
```

El ejemplo anterior es solo un programa sencillo, que crea dos variables, 'a' y 'b', que tienen como dominio $[1, 2]$, que crea una restricción para que el valor de 'a' sea diferente del de 'b', y que después recupera las soluciones del problema. En las siguientes secciones se explicará en detalle cómo se pueden construir programas más complejos, tomando como referencia el ejercicio de los alumnos que tienen que elaborar un trabajo de Ingeniería del Software. Dicho ejercicio se puede encontrar en las hojas de ejercicios subidos a Aula Global. [1]

3 Creación de variables

Existen diferentes formas de crear las variables mediante las funciones *addVariable* y *addVariables* que ofrece la librería. Algunos ejemplos serían:

```
# Crea la variable 'a' que tiene como dominio [1, 2]
problem.addVariable('a', [1, 2])
# Crea las variables 'a' y 'b', ambas con el dominio [1, 2, 3]
problem.addVariables("ab", [1, 2, 3])
# Crea las variables 'a' y 'b', ambas con el dominio [0, 1, 2]
problem.addVariables(['a', 'b'], range(3))
```

Para el problema del grupo de alumnos que tienen que elaborar un documento de Ingeniería del Software, tenemos seis variables (J, M, A, Y, R, F), donde cada una se corresponde con un alumno diferente, y todas tienen como dominio $[1, 2, 3]$ salvo Juan, puesto que el enunciado dice que no tiene conocimientos para hacer la primera parte, luego su dominio sera $[2, 3]$, y María, que solo quiere trabajar en la tercera parte, luego su dominio será $[3]$. Teniendo en cuenta ésto, la sección de código que nos permite crear estas variables junto con sus dominios sería la siguiente:

```
problem.addVariables(['A', 'Y', 'R', 'F'], [1, 2, 3])
```

[1] También es posible consultar más ejemplos en `https://labix.org/python-constraint` o `https://stackabuse.com/constraint-programming-with-python-constraint/`

```
problem.addVariable('J', [2, 3])
problem.addVariable('M', [3])
```

4 Creación de restricciones

Una vez definidas las variables de nuestro problema, se añaden las restricciones mediante la función *addConstraint* proporcionada por la librería. Antes de continuar con el ejercicio de los alumnos, algunos ejemplos de utilización de esta función serían los siguientes:

```
problem.addConstraint(lambda a, b: a > b, ('a', 'b'))
```

En el ejemplo anterior, se crea una función *lambda* que recibe dos parámetros que se corresponden con los valores de las variables 'a' y 'b', y comprueba que 'a' es mayor que 'b'. Se puede comprobar que para operaciones sencillas se puede hacer uso de la función *lambda* que proporciona *python*. Una forma alternativa de hacer lo mismo que en el caso anterior sería:

```
def greater(a, b):
        if a > b:
                return True

problem.addConstraint(greater, ('a', 'b'))
```

Es decir, primero se crea una función llamada *greater* que comprueba que la variable 'a' es mayor que la variable 'b'. En la función *addConstraint* simplemente se invoca esta función auxiliar que hemos creado, indicando, además, los parámetros que va a recibir. La función *greater* se invocará tantas veces como posibles pares de valores de 'a' y de 'b' existen, y devolverá verdadero siempre que sea un par válido, es decir, siempre que 'a' sea mayor que 'b', y falso en caso contario. Otro ejemplo de restricción sería:

```
def notEqual(*args):
        for i in range(len(args)):
                for j in range(i+1, len(args)):
                        if i != j and args[i] == args[j]:
                                return False
        return True

problem.addConstraint(notEqual, ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'))
```

En este último ejemplo, se comprueba que los valores de las variables comprendidas entre 'a' y 'h' es diferente. Se puede comprobar que el parámetro recibido por la función *notEqual* en este caso es un array, $*args$, que contiene el valor de las ocho variables que se están considerando. Utilizando este array evitamos tener que declarar la función *notEqual* con ocho parámetros. Nuevamente, la función *notEqual* se invocará tantas veces como posibles valores puedan tomar las variables, y se descartarán aquellos que sean inválidos según el criterio considerado.

Volviendo al ejercicio que nos ocupa, vamos a modelar en primer lugar la restricción de que Alfredo y Rubén no quieren trabajar juntos, es decir, $R_{A,R} = \{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)\}$. Se puede hacer de varias formas. Una de ellas es definir una función, por ejemplo, *notEqual*, similar a la anterior, que comprueba que el valor de la variable 'A' es diferente al de la variable 'R':

```
def notEqual(a, b):
        if a != b:
                return True

problem.addConstraint(notEqual, ('A', 'R'))
```

Dada la simplicidad de la función, esta misma restricción también se puede modelar mediante una función lambda:

```
problem.addConstraint(lambda a, b: a != b, ('A', 'R'))
```

Por ultimo, la librería ofrece la función *AllDifferentConstraint* que precisamente comprueba que el valor de una variable es diferente al de las otras:

```
problem.addConstraint(AllDifferentConstraint(), ['A', 'R'])
```

Lo anterior, son tres formas diferentes de modelar la misma restriccion $R_{A,R} = \{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)\}$. Ahora modelamos la restricción de que Rubén y Felisa quieren trabajar en la misma parte $R_{R,F} = \{(1,1),(2,2),(3,3)\}$

```
problem.addConstraint(lambda a, b: a == b, ('R', 'F'))
```

Por último, modelamos la restricción de que Yara debe hacer una parte posterior a la de Rubén, $R_{R,Y} = \{(1,2),(1,3),(2,3)\}$.

```
def consecutive(a, b):
        if b > a:
                return True

problem.addConstraint(consecutive, ('R', 'Y'))
```

En este caso, se ha creado la función *consecutive* que comprobará para todos los pares de valores de las variables 'R' y 'Y', cuáles son válidos, en cuyo caso la función devolverá verdadero, y cuáles son inválidos, en cuyo caso la función devolverá falso.

Como se ha podido comprobar la librería ofrece funciones predefinidas para construir las restricciones. Algunas de estas funciones son:

- **AllDifferentConstraint**: Fuerza a que el valor de las variables indicadas sea diferente. Si no se indican las variables, se asume que son todas las variables.
- **AllEqualConstraint**: La contraria a la anterior. Fuerza a que todas las variables indicadas tomen el mismo valor.
- **MaxSumConstraint**: Fuerza a que la suma de las variables indicadas resulte como máximo en un determinado valor.
- **ExactSumConstraint**: Fuerza a que la suma de las variables indicadas resulte exactamente en el valor indicado.
- ...

5 Recuperación de la solución

Una vez que se ha construido el modelo, se recupera la solución o soluciones que satisfacen las restricciones, es decir, se recuperan los valores que pueden tomar cada una de las variables que hacen que se cumplan todas las restricciones impuestas. Para este cometido, la librería ofrece dos funciones diferentes: *getSolution*() y *getSolutions*(). La primera recupera una única solución, mientras que la segunda recupera todas las posibles soluciones que satisfacen las restricciones.

Finalmente, el programa completo que modela el ejercicio de los alumnos y su trabajo de Ingeniería del Software sería el que se muestra a continuación [2]:

```
from constraint import *

problem = Problem()

problem.addVariables(['A', 'Y', 'R', 'F'], [1, 2, 3])
problem.addVariable('J', [2, 3])
problem.addVariable('M', [3])

problem.addConstraint(AllDifferentConstraint(), ['A', 'R'])
problem.addConstraint(lambda a, b: a == b, ('R', 'F'))
def consecutive(a, b):
        if b > a:
                return True
problem.addConstraint(consecutive, ('R', 'Y'))

print(problem.getSolution())
```

[2] También puede encontrarse en Aula Global con el nombre *alumnos.py*