MANUEL GUERRERO MOÑÚS

ALEJANDRO CILLEROS GARRUDO

FACULTAD DE INFORMÁTICA UCM | 2018 - 2019

INTRODUCCIÓN

¿Qué es JaCoP?:

- Es una librería Java para modelar y resolver problemas mediante el paradigma de programación con restricciones.
- Permite trabajar con dominios de enteros, reales y booleanos.

Descarga e instalación:

- Dirección web: https://sourceforge.net/projects/jacop-solver/
- Para usarlo solamente es necesario importar la librería en un proyecto Java.

MATEMÁTICO VS COMPUTACIONAL

Realidad		JaCop Software
Modelo matemático	→	Store
Variables	→	IntVar, FloatVar, BooleanVar , SetVar
Restricciones	→	No/Primitive, Globals, Conditionals, Logicals
Resolución del modelo	→	Search

MODELO MATEMÁTICO: STORE

- La variable Store almacenará la definición formal de muestro problema.
- Es lo primero que hay que inicializar ya que todas las restricciones y variables estarán vinculadas a él.
- ¡¡¡ NO OLVIDAR !!!

```
Store store = new Store();
```

VARIABLES: IntVar, FloatVar, BooleanVar, SetVar

- IntVar x = new IntVar(store , “x” , 1 , 100);
  - store → Modelo matemático.
  - “x” → Nombre / Identificador de la variable dentro del modelo.
  - 1 → Cota inferior de la variable.
  - 100 → Cota superior de la variable.
    - x.addDom(120 , 160); → Variable Multidominio.
- FloatVar y = new IntVar(store , “y” , 1.0 , 100.0);
  - FloatDomain.setPrecision(<int>); → Precisión decimal.
- BooleanVar z = new BooleanVar(store , “z”);
- SetVar *s* = new SetVar(store , “s” , 1 , 3);

RESTRICCIONES:

Tipos de restricciones:
1.- Primitive/No Primitive
2.- Logical/Set
3.- Globals

A su vez pueden ser:

A) **Hard Constraint**: Aquellas que siempre han de cumplirse.

- store.impose(<constraint>);

B) **Soft Constraints**: Aquellas que pueden o no cumplirse. Requieren reificar.

- store.impose(new Reified(<constraint>,<Boolean>));

1.- Primitive and NonPrimitive Constraints

Primitive Constraint	JaCoP specification
$X = Const$	XeqC(X, Const)
$X = Y$	XeqY(X, Y)
$X \neq Const$	XneqC(X, Const)
$X \neq Y$	XneqY(X, Y)
$X > Const$	XgtC(X, Const)
$X > Y$	XgtY(X, Y)
$X \geq Const$	XgteqC(X, Const)
$X \geq Y$	XgteqY(X, Y)
$X < Const$	XltC(X, Const)
$X < Y$	XltY(X, Y)
$X \leq Const$	XlteqC(X, Const)
$X \leq Y$	XlteqY(X, Y)
$X \cdot Const = Z$	XmulCeqZ(X, Const, Z)
$X + Const = Z$	XplusCeqZ(X, Const, Z)
$X + Y = Z$	XplusYeqZ(X, Y, Z)
$X + Y + Const = Z$	XplusYplusCeqZ(X, Y, Const, Z)
$X + Y + Q = Z$	XplusYplusQeqZ(X, Y, Q, Z)
$X + Const \leq Z$	XplusCleqZ(X, Const, Z)
$X + Y \leq Z$	XplusYleqZ(X, Y, Z)
$X + Y > Const$	XplusYgtC(X, Y, Const)
$X + Y + Q > Const$	XplusYplusQgtC(X, Y, Q, Const)
$\|X\| = Y$	AbsXeqY(X, Y)

Non-primitive Constraint	JaCoP specification
$X \cdot Y = Z$	XmulYeqZ(X, Y, Z)
$X \div Y = Z$	XdivYeqZ(X, Y, Z)
$X \mod Y = Z$	XmodYeqZ(X, Y, Z)
$X^Y = Z$	XexpYeqZ(X, Y, Z)

Ejemplos:

```
IntVar X = new IntVar( store , "x" , 1 , 100 );
IntVar Y = new IntVar( store , "y" , 1 , 100 );
```

- `store.impose(new XeqY(X,Y));`
- `store.impose(new XplusYeqZ(X,Y,Z));`

2.- Logical Constraints & Set Constraints

Primitive Constraint	JaCoP specification
$\neg c$	`Not(c);`
$c1 \vee c2 \vee \cdots \vee cn$	`PrimitiveConstraint[] c = {c1, c2, ...cn};` `Or(c);`
$c1 \wedge c2 \wedge \cdots \wedge cn$	`PrimitiveConstraint[] c = {c1, c2, ...cn};` `And(c);`
$c1 \Leftrightarrow c2$	`Eq(c1, c2);`
X in Dom	`In(X, Dom);`
$c \Leftrightarrow B$	`Reified(c, B);`
$c \Leftrightarrow \neg B$	`Xor(c, B);`
if $c1$ then $c2$	`IfThen(c1, c2);`
if $c1$ then $c2$ else $c3$	`IfThenElse(c1, c2, c3);`

Primitive Constraint	JaCoP specification
$e \in A$	`EinA(e, A)`
$S_1 = S_2$	`AeqB(S1, S2)`
$S_1 \subseteq S_2$	`AinB(S1, S2)`
Non-primitive Constraint	**JaCoP specification**
$S_1 \bigcup S_2 = S_3$	`AunionBeqC(S1, S2, S3)`
$S_1 \bigcap S_2 = S_3$	`AintersectBeqC(S1, S2, S3)`
$S_1 \setminus S_2 = S_3$	`AdiffBeqC(S1, S2, S3)`
$S_1 <> S_2$	`AdisjointB(S1, S2)`
Match	Match(Set, VarArray)
$\#S = C$	`CardA(S, C)`
$\#S = X$	`CardAeqX(S, X)`
Weighted sum < S, W > = X	`SumWeightedSet(S, W, X)`
$Set[X] = Y$	`ElementSet(X, Set, Y)`

Ejemplos:

```
Constraint C = new XeqY(X,Y);

store.impose(new Not(C));

store.impose(new IfThenElse(C1,C2,C3));
```

Ejemplos:

```
SetVar S = new SetVar( store , "s" , 8 , 11 );
Constraint C = new AunionBeqC(S1,S2,S3);

store.impose(C);
```

3.- Global Contraints

used with ==, <, >, <=, >=, !=

$x_1 + x_2 + \cdots + x_n = sum$

```
IntVar[] x = {x1, x2, ..., xn};
IntVar sum = new IntVar(...)
SumInt(store, x, "==", sum);
```

alldifferent$([x_1, x_2, \ldots, x_n])$

```
IntVar[] x = {x1, x2, ..., xn};
Alldifferent(x);
```

circuit$([x_1, x_2, \ldots, x_n])$

```
IntVar[] x = {x1, x2, ..., xn};
Circuit(x);
```

count$([x_1, x_2, \ldots, x_n], var, value)$

```
int value = ...;
IntVar var = new IntVar(...);
IntVar[] x = {x1, x2, ..., xn};
Count(x, var, value);
```

atLeast$([x_1, x_2, \ldots, x_n], var, value)$

```
int value = ...;
int minCount = ...;
IntVar[] x = {x1, x2, ..., xn};
AtLeast(x, minCount, value);
```

cumulative$([t_1, t_2, \ldots, t_n], [d_1, d_2, \ldots, d_n], [r_1, r_2, \ldots, rn], ResourceLimit)$

```
IntVar[] t ={t1, t2, ..., tn};
IntVar[] d ={d1, d2, ..., dn};
IntVar[] r = {r1, r2, ..., rn};
IntVar Limit = new IntVar(...);
Cumulative(t, d, r, Limit);
```

lex$([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_m])$

```
IntVar[] x = {{x0, x1 ..., xn};
IntVar[] y = {{y0, y1 ..., yn};
LexOrder(x, y);
```

regular$(fsm, [x_1, x_2, \ldots, x_n])$

```
FSM fsm = new FSM();
IntVar[] x = {x1, x2, ..., xn};
Regular(fsm, x);
```

```
IntVar d = new IntVar[N];
Constraint ct = new Circuit(d);

store.impose(ct);
```

RESOLUCIÓN DEL MODELO: SEARCH

1º - **Eligiendo nuestra búsqueda:**

Para buscar soluciones por defecto se usa la búsqueda en profundidad y solo se busca una única solución.

```
Search<Type> search = new DepthFirstSearch<Type>();
```

2º - **Eligiendo nuestra estrategia:**

```
SelectChoicePoint<Type> strategy = new InputOrderSelect<Type>( Store , Array , Value Strategy );
SelectChoicePoint<FloatVar> strategy = new SplitSelectFloat<FloatVar>( Store , Array , Value Strategy );
SelectChoicePoint<Type> strategy = new RandomSelect<Type>( Array , Value Strategy );
SelectChoicePoint<Type> strategy = new TraceGenerator<Type>( Search , Strategy , Array );
SelectChoicePoint<Type> strategy = new SimpleSelect<Type>( Array , Variable Strategy , Value Strategy );
SelectChoicePoint<Type> strategy = new SimpleMatrixSelect<Type>( Matrix , Variable Strategy , Value Strategy );
```

Value Strategy:

* **IndomainMin**: Selecciona el valor mas pequeño del dominio de la variable.
* **IndomainMax:** Selecciona el valor mas grande del dominio de la variable.
* **IndomainMiddle:** Selecciona el valor intermedio del dominio de la variable.
* **IndomainRandom:** Selecciona un valor aleatorio del dominio de la variable.

Variable Strategy:

* **SmallestDomain:** Se selecciona cada vez la variable con el dominio mas pequeño.
* **MostConstrainedStatic:** Selecciona cada vez la variable que mas restricciones tiene asociadas.
* **SmallestMin:** Selecciona cada vez la variable que tiene el menor valor en su dominio.
* **LargestDomain:** Se selecciona cada vez la variable con el dominio mas grande.
* **LargestMin:** Selecciona cada vez la variable con el mayor valor de su dominio.

Configuración de la búsqueda:

Search Methods:

* setTimeOut(<int>);

- La búsqueda de soluciones se interrumpe al cabo de los segundos especificados si no se ha encontrado ninguna solución.

* setPrintInfo(false);

- No muestra información de la búsqueda.

* getSolutionListener().searchAll(true);

- Busca todas las soluciones posibles.

* getSolutionListener().recordSolutions(true);

- Almacena las soluciones encontradas.

* getSolutionListener().solutionsNo();

- Obtiene el número de soluciones que se han podido encontrar.

* getSolution(<int>)[<int>]

- Acceder a un campo de una solución. (Requiere almacenar soluciones)

Consistencia y ejecución:

```
if(store.consistency()) {    // Se estudiará si el modelo no presenta ningún error lógico.
        // Aquí podremos configurar algunas propiedades de la búsqueda con el objeto Search.
        if(search.labeling( store , strategy [ , ± cost ] )) {  // ¿ Es posible solucionar el modelo ?
                System.out.println("Existe solucion.");
                // Aquí mostraremos el valor de nuestras variables, podemos usar el objeto Search.
        } else {
                System.out.println(“No existe solucion.");
        }
} else {
        System.out.println("Modelo inconsistente.");
}
```

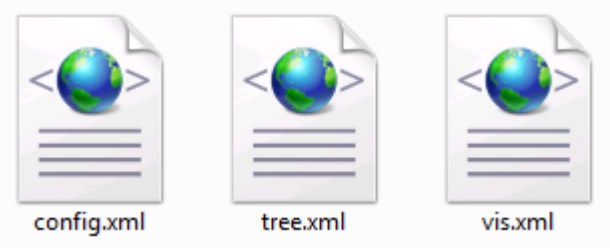
Herramienta CPviz:

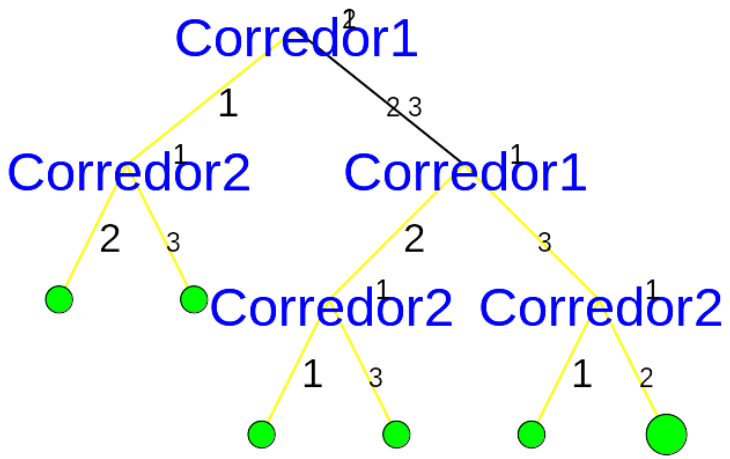
Cpviz es una herramienta que permite mostrar de forma gráfica la traza de la ejecución de un programa. Requiere el uso del objeto TraceGenerator.

```
Search<Type> search = new DepthFirstSearch<Type>();

SelectChoicePoint<Type> strategy = new SimpleSelect<Type>(Array,Variable Strategy,Value Strategy);

SelectChoicePoint<Type> trace = new TraceGenerator<Type>(search,strategy,Array of vars to trace);

search.labeling( store , trace [ , ± cost ] );
```

Ficheros de configuración:

- config.xml: Contiene información de como se han de dibujarse los gráficos. Se configura a mano.
- tree.xml y vis.xml: Contienen toda la información del árbol de búsqueda sobre el que se ha realizado el recorrido para poder dibujarlo. Estos ficheros se generan automáticamente al ejecutarse el programa.

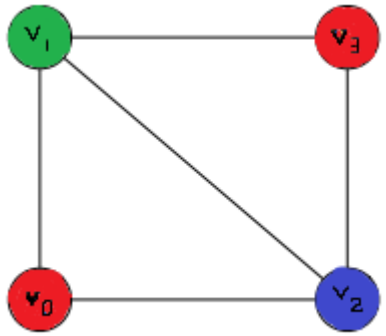




Un ejemplo básico

```
public class Main {

    public static void main (String[] args) {
        Store store = new Store();
        int size = 4;

        //  define finite domain variables
        IntVar[] v = new IntVar[size];
        for(int i = 0; i < size; i++)
            v[i] = new IntVar(store, "v"+i, 1, size);

        // define constraints
        store.impose(new XneqY(v[0], v[1]));
        store.impose(new XneqY(v[0], v[2]));
        store.impose(new XneqY(v[1], v[2]));
        store.impose(new XneqY(v[1], v[3]));
        store.impose(new XneqY(v[2], v[3]));

        // search for a soluction and print results
        Search<IntVar> search = new DepthFirstSearch<IntVar>();
        SelectChoicePoint<IntVar> select = new InputOrderSelect<IntVar>(store, v,
                                                    new IndomainMin<IntVar>());

        boolean result = search.labeling(store, select);

        if(result)
            System.out.println("Solution: " + v[0] + ", " + v[1] + ", " +
                                              v[2] + ", " + v[3]);
        else
            System.out.println("No hay solucion");

    }
}
```

```
Labeling has finished with return value of true
Depth First Search DFS1
[v0 = 1, v1 = 2, v2 = 3, v3 = 1]

Solution : [v0=1, v1=2, v2=3, v3=1]
Nodes : 4
Decisions : 4
Wrong Decisions : 0
Backtracks : 0
Max Depth : 4

Solution: v0 = 1, v1 = 2, v2 = 3, v3 = 1
```



Ventajas e inconvenientes observados

Ventajas:

* Ya que todas las operaciones existentes están dentro de objetos, tenemos un mayor conocimiento de lo hecho y una probabilidad mas reducida de que nos equivoquemos, al menos a nivel sintáctico.

* Incrementa el potencial de un lenguaje de programación completo como lo es Java.

* Soluciones muy detalladas, además de numerosas posibilidades de exploración estas.

Inconvenientes:

* Incompatibilidad con los tipos de Java.

* No hay variables constantes. Tenemos que usar restricciones para fijar el valor.