PRÁCTICA 2:
PROBLEMAS DE SATISFACCIÓN DE RESTRICCIONES.
SESIÓN 1

Técnicas de los Sistemas Inteligentes
Curso 2015-2016
3er. Curso Grado en Informática

5. Para una línea ferroviaria, Renfe dispone de 4 trenes (T_1, T_2, T_3 y T_4) y 3 locomotoras (L_1, L_2 y L_3). El horario diario en el que tienen que circular los trenes es:

Tren	Horario
T1	8 a 10
T2	9 a 13
T3	12 a 14
T4	11 a 15

Además, hay que tener en cuenta lo siguiente:

- Cada locomotora sólo puede tirar de un tren cada vez.
- Con una hora, cada locomotora dispone de tiempo para estar en la estación preparada para el próximo tren.
- L_3 no tiene potencia para arrastrar a T_3.
- L_2 y L_3 no tienen potencia para arrastrar a T_4.

Se desea saber que distribución hay que realizar para que puedan circular todos los trenes en sus respectivos horarios. Para ello, se pide:

a) Plantear el problema como un PSR de dos formas diferentes: tomando como variables los trenes y tomando como variables las locomotoras. Analizar qué representación es más adecuada, razonando la respuesta.

b) Con la representación elegida, resolver el problema usando búsqueda con vuelta atrás y las heurísticas que considere más apropiadas para

Objetivo

- Conocer un nuevo paradigma de programación
  - Programación lógica de restricciones
    - Basada en Prolog
    - Definición de un CSP: variables, dominios y restricciones, basada en un lenguaje declarativo.
    - Permite centrarse en el modelado de problemas de satisfacción de restricciones, en lugar de implementar la estrategia de búsqueda.
    - Proporciona librerías de funciones para declarar procesos de búsqueda
- Conocer ejemplos de CSPs clásicos.
- Resolver CSPs siguiendo este paradigma
  - Modelar CSP (vars, domains, constraints)
  - Implementar programas declarativos que resuelven CSPs.

Cómo conseguirlo

- Tenemos que llegar a entender cómo representar y modelar CSPs
  - y resolverlos con técnicas de búsqueda y heurísticas de propósito general.
- Pero antes hay que refrescar Prolog
  - Un lenguaje declarativo basado en lógica de predicados de primer orden
  - Tiene implícito un mecanismo de búsqueda con vuelta atrás.

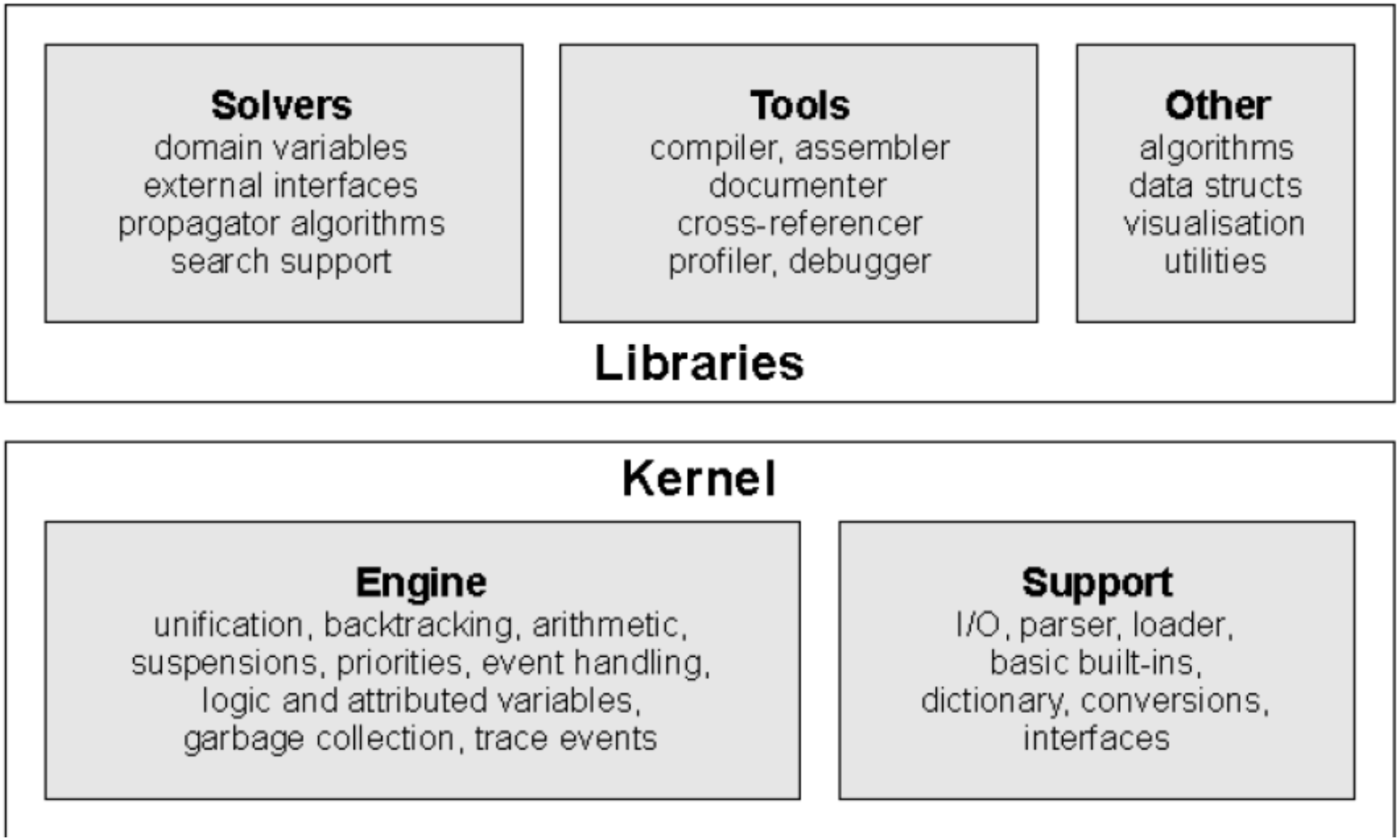
Esquema

- Eclipse CLP
- Prolog en 30 min.
  - Básicos
  - Backtracking en Prolog
  - Artimética y comparación en Prolog
  - Esquema simple de CSP en Prolog
  - Listas en Prolog
- CSP en Eclipse
  - Esquema básico de CSP en Eclipse
  - Iteración en Eclipse
  - Propagación en Eclipse
  - Librerías de Eclipse
  - Librería sd: coloreado de mapas
  - Librería ic: puzzle criptoaritmético
  - Restricciones globales
  - Labeling

Eclipse Constraint Programming System

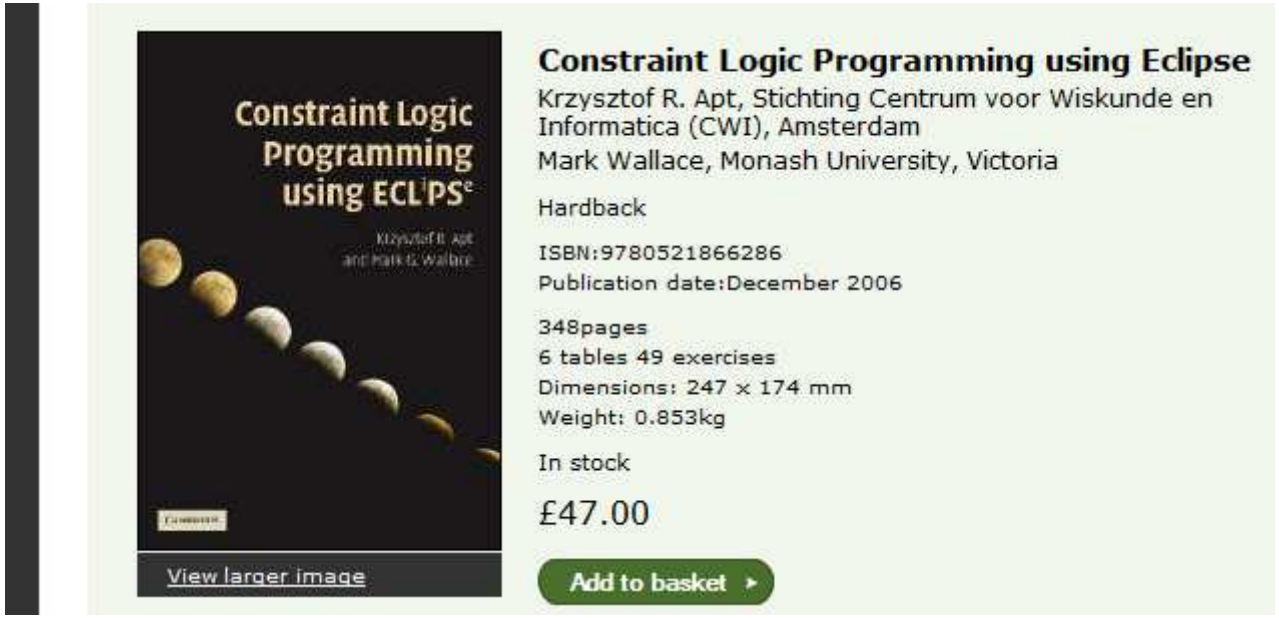
- http://eclipseclp.org/index.html
- Consiste en
  - **un motor** para la interpretación en tiempo de ejecución de programas de restricciones
  - una colección de **librerías** para manejar distintos tipos de CSPs (discretos, contínuos,...)
  - **un lenguaje** para el modelado de restricciones con estructuras de control que **extienden Prolog**
  - un **entorno** de desarrollo
  - interfaces para **integrar** en distintas aplicaciones
  - interfaces para **resolutores** (solvers) de CSPs de terceros
- Características
  - Tareas generales de programación, **prototipado rápido**
  - Resolución de problemas usando las librerías que propociona basadas en el paradigma CLP: Constraint Logic Programming
  - Desarrollo de nuevos resolutores de problemas basados en los “solvers” que proporciona en las características de bajo nivel de Eclipse.

Arquitectura de Eclipse CLP



Eclipse información y documentación

Constraint Logic Programming using ECLiPSe



http://eclipseclp.org/doc/

[1]J. Schimpf y K. Shen, «ECLiPSe - from LP to CLP», *arXiv:1012.4240*, dic. 2010.

- Antoni Niederlinski , "A Gentle Guide to Constraint Logic Programming *via* ECLiPSe", Third edition, 2014, 570 p. ISBN 978-83-62652-08-2. http://www.anclp.pl/ (En la biblioteca).

Eclipse información y documentación

- Curso e-learning por Helmut Simonis
  - http://4c.ucc.ie/~hsimonis/ELearning/
- Página de ejemplos de Hakan Kjellerstrand
  - http://www.hakank.org/eclipse/
- Guía on-line sobre Programación de Restricciones
  - Genérica, no centrada en ECLiPSe
  - http://kti.ms.mff.cuni.cz/~bartak/constraints/

- Un programa Prolog se construye a partir de **términos:**
  - Átomos: un átomo es una constante no numérica (*constante simbólica*), representada por una secuencia de caracteres empezando por **minúscula** o **delimitada por “”.** `blu_sky, “Juanito”`
  - Variables: secuencia de caracteres empezando por **mayúscula o subrayado.** `X, Juanito, _quien, _cuanto`
    - **No son huecos de memoria que rellenar** como en paradigmas no declarativos.
    - **Interpretarlas como incógnitas,** igual que en lógica o álgebra.
    - Variable especial: **_ (subrayado)** es una variable anónima cuyo contenido no interesa.
  - Números: constantes enteras o de punto flotante

- **Predicados**: una **relación** entre variables, `gustar(Alguien, Algo)`
  - `<nombre><argumentos>`
  - Aridad: número de argumentos en predicados.
    - Referencia de predicado **gustar/2.**
  - **Significado:** para que una relación tenga sentido hay que definir dominios para las variables (p.ej. Alguien toma valores en nombres de estudiantes, Algo toma valores en nombres de lenguajes de programación).
  - "A Alguien le gusta Algo" -> gustar(Alguien,Algo)
    - La notación prefija permite expresar muchos argumentos de forma más cómoda que el lenguaje natural.
    - Un predicado por sí mismo no tiene valor lógico, es ambiguo.
  - Es un patrón para un **predicado instanciado** con variables ligadas a constantes: `gustar("Juanito","PROLOG").` De esta forma puede ser **cierto** o **falso**
    - El orden de los argumentos importa
    - Predicados anidados: los argumentos pueden ser predicados: `gustar(estudiante(Alguien),lenguaje(Algo))`
    - **Funciones:** predicados especiales que devuelve un valor numérico `+(X1,X2) -> X1 + X2`
  - **Tipos de predicados:**
    - **Internos (built-in, standard)**
      - Ver manual de referencia de ECLiPse: http://www.eclipseclp.org/doc/bips/index.html
    - **Privados:** los definidos por el programador

- **Estructuras**: representan una tupla de un número fijo de átomos. <nombre><argumentos>, representación similar a un predicado: nombre, argumentos,aridad. El nombre de una estructura se llama ***functor.***
  - ***No contienen variables.***
  - Usadas para representar datos:
  - registroBD("Juan Fernández",26756483S,600934679,varon,…)
- **Listas**
  - [a,b,"CDE", 5, F],
  - lista vacía []
  - Más adelante las vemos con algo más de detalle

Programa Prolog

- **Hechos**
  - Hecho: predicado instanciado, sus argumentos son constantes
  - Variables en mayúscula
  - Constantes en minúscula
- **Reglas**
- Escritos en lógica de predicados de primer orden

Hechos

```
%homer es un varón
%marge es una hembra
varon(homer).
hembra(marge).
%instancias de los
  predicados varon(X) y
  hembra(X)
%predicados unarios,
  expresan propiedades de
  un objeto, representado
  como una constante
```

Ejemplos en: http://decsai.ugr.es/~faro/TSI/ejemplosPracticaCSP.html

Programa Prolog

- Hechos
  - Hecho: predicado instanciado, sus argumentos son constantes
  - Variables en mayúscula
  - Constantes en minúscula
- Reglas
- Escritos en lógica de predicados de primer orden

Hechos

```
varon(homer).
hembra(marge).
padre(homer,lisa).
madre(marge, lisa).
madre(marge, bart).
padre(abe, homer).
padre(abe, herbert).
padre(homer, bart).
%predicados n-arios,
  expresan relaciones
  entre objetos.
```

Programa Prolog

- Hechos
- Reglas
  - implicaciones lógicas formadas por
    - un antecedente (body, cuerpo)
    - y un consecuente. (head, cabecera)
  - La cabecera es un único término.
  - El cuerpo es habitualmente una **conjunción lógica**
  - Se usa ',' *(operador AND)* para separar los componentes **(términos)** del cuerpo.
  - Una regla se denomina cláusula.
- Escritos en lógica de predicados de primer orden

Reglas: conjunción

```
% X e Y son hermanos
  SI  X es distinto de
  Y y tienen el mismo
  padre.

hermano(X,Y) :-
  padre(Z,X),
  padre(Z,Y), X\=Y.
```

Programa Prolog

- Hechos
- Reglas
  - implicaciones lógicas formadas por
    - un antecedente (body, cuerpo)
    - y un consecuente. (head, cabecera)
  - La cabecera es un único término.
  - El cuerpo es habitualmente una conjunción lógica
  - Se usa ',' para separar los componentes **(términos)** del cuerpo.
  - Una regla se denomina cláusula.
- Escritos en lógica de predicados de primer orden

Reglas: disyunción

```
% Si X es padre de Y, entonces
  X es ascendiente directo de
  Y.
% Si X es madre de Y, entonces
  X es ascendiente directo de Y

ascendientedirecto(X,Y) :-
  padre(X,Y).
ascendientedirecto(X,Y) :-
  madre(X,Y).

%% o de forma alternativa

ascendientedirecto(X,Y) :-
  padre(X,Y);madre(X,Y).
```

En resumen

```
%Datos :: Base de hechos.
varon(homer).
hembra(marge).
padre(abe, homer).
padre(abe, herbert).
padre(homer, bart).
padre(homer,lisa).
madre(marge, lisa).
madre(marge, bart).

%Programa: clásulas
hermano(X,Y) :-
  padre(Z,X), padre(Z,Y), X\=Y.


ascendientedirecto(X,Y) :-
   padre(X,Y);madre(X,Y).
```

- Podemos ver un programa Prolog como un conjunto de procedimientos, donde cada uno se representa como un predicado.
- Cada predicado se define a partir de un conjunto de cláusulas.
- Cláusula es una estructura que acaba en “.”
- En general nos referimos con cláusula a una implicación lógica, pero puede ser un predicado instanciado de la base de conocimiento.
- Una cláusula puede interpretarse como una implicación, la coma separa objetivos y se interpreta como conjunción.
- Si hay varias cláusulas para un mismo predicado, lo interpretamos como una disyunción entre cláusulas (todas especifican formas distintas de llevar a cabo un procedimiento)
- El ámbito de una variable se extiende sobre la cláusula en la que está.
- Variables en diferentes cláusulas son diferentes, aunque tengan el mismo nombre.

Lenguaje Orientado a Objetos

Clase Persona

- Varon:boolean
- Hembra:boolean
- Edad:integer
- Padre: Persona
- Madre:Persona

Programación Lógica

- `Persona(homer)`
- `Varon(homer)`
- `Edad(homer, 50)`
- `Padre(homer,abe)`
- `Persona(lisa)`
- `Hembra(lisa)`
- `Edad(lisa,10)`
- `Padre(lisa,homer)`
- `...`

A continuación ...

1. Activación de un programa Prolog
2. Unificación
3. Definiciones recursivas
4. Mecanismo de Búsqueda y backtracking en Prolog
5. Artimética en Prolog
6. Comparación en Prolog
7. Uso de fallo (backtracking manual)
8. Esquema simple para resolver CSPs

Ejecución/activación de una programa Prolog

- Un programa Prolog se activa mediante una consulta:
  - predicado (parcial o totalmente instanciado)
  - del que queremos saber su valor de verdad
  - respecto a la base de conocimiento que forma el programa
- El predicado usado en la consulta es también denominado meta u objetivo
- El mecanismo de ejecución de Prolog trata de determinar su certeza mediante la aplicación de las reglas (en el orden en que se han escrito).

- Las consultas pueden usarse bien para comprobar
  - ?:- madre(homer,bart)
  - ?:- hermano(lisa,bart)
- o para encontrar una respuesta
  - ?:- padre(X,bart)
- o varias respuestas
  - ?:- hermano(X,bart).

Unificación

- Para encontrar respuesta a una consulta, Prolog realiza una búsqueda en la base de conocimiento .
- Trata de emparejar el predicado de la consulta con los hechos y reglas.
- Para emparejar predicados Prolog usa **unificación:**
  - tratar de encontrar sustituciones de variables
  - para que dos términos sean iguales.

- marge y marge unifican porque son el mismo término.
- marge y X unifican porque X la podemos instanciar con marge para que ambos términos sean iguales
- hembra(marge) y hembra(X) unifican por que la propiedad tiene el mismo nombre y X la podemos instanciar con marge, lo que da como resultado dos términos iguales
- ....

Definiciones recursivas

- **Backtracking.**
  - El mecanismo de ejecución de Prolog está basado en una búsqueda primero en profundidad con backtracking.

```
ancestro(X,Y) :-
   ascendientedirecto(X,Y).
ancestro(X,Y) :-
   ascendientedirecto(Z,Y),
   ancestro(X,Z).
```

```
Ejecuta(consulta)
   IF consulta = vacío RETURN Success
   FOR EACH objetivo IN consulta
   (se recorren de izda a dcha)
      Punto de Decisión: Clausulas = lista de clausulas
          H:-B tales que H unifica objetivo
      (ordenadas por orden de escritura en el código)
      IF clausulas = vacío RETURN Fail
      FOR EACH clausula
         Aplicar Unificación
         Result = Ejecuta(Body(clausula))
         IF Result = Success
         THEN RETURN Success
         ELSE Continue
      RETURN Fail (ninguna clausula devuelve Success)
   RETURN Fail (ningún objetivo devuelve Success)
```

Búsqueda y Backtracking en PROLOG

- El proceso de cálculo de Prolog está basado en el algoritmo de **Resolución**
- Dado un conjunto de hechos y reglas (programa)
- La ejecución cominenza con una consulta: objetivo inicial a ser resuelto.
  - El conjunto de objetivos que tienen que ser resueltos se llama *resolvente*.

1. **goal** = resolvente.pop()
   1. Si **goal** es vacío FIN.
2. **Alternativas = cláusulas** cuyo **head** unifica **goal** .
   Si **Alternativas es vacía, GOTO 6.**
3. **Punto de decisión**:
   clausula = Alternativas.pop()
   (Alternativas) Si hay más cláusulas, recordar las restantes.
4. Unificar **goal** con **head(clausula)** (instanciación de variables en el goal y en la cláusula).
5. **Resolvente.push(body(claúsula)). GOTO 1.**
6. Backtracking: ir al más reciente punto de decisión. **GOTO 3.**

```
ancestro(X,Y) :- padre(X,Y).                    %clause 1
ancestro(X,Y) :- padre(Z,Y), ancestro(X,Z).     %clause 2
padre(abe, homer).                              % clause 3
padre(abe, herbert).                            % clause 4
padre(homer, bart).                             % clause 5
padre(marge, bart).                             % clause 6
```

Aritmética en PROLOG

- Operadores predefinidos para aritmética básica:
  - +,-,*,div,mod,^, -(unario),abs
- Si no se especifica lo contrario, los operadores son como cualquier otra relación (predicado)
  - X = 1+2.
    Respuesta: X=1+2 . X unificado al término +(1,2)
- Operador "is": obliga a la evaluación
  - X is 1+2
    Respuesta: X = 3
- A is B
  - Evaluar B a un número y unificar el resultado con A.
- Los operadores de comparación fuerzan también la evaluación
  - 145 * 34 > 100.
    Respuesta: Yes.

Comparación en PROLOG

Operadores de comparación

X > Y	X es mayor que Y
X < Y	X es menor que Y
X >= Y	X es mayor o igual que Y
X =< Y	X es menor o igual que Y
X =:= Y	Los valores de X e Y son iguales
X =\= Y	Los valores de X e Y no son iguales

- Ejemplos:
  - 1 + 2 =:= 2 + 1.
    > Yes
  - 1 + 2 = 2 + 1.
    > No
  - 1 + A = B + 2.
    > A = 2
    > B = 1
  - 1+A =:= B+2.
    >Error: Variable no instanciada en expresión aritmética.

= versus =:= versus "is"

- **X=Y** causa la unificación de X e Y y posiblemente instanciación de variables
- **X =:= Y** causa una
  - evaluación aritmética de X e Y
  - No puede dar lugar a instanciación de variables
- **X is Y** causa una
  - Evaluación aritmética de Y
  - Unificación de X con el resultado de evaluar Y

- Ejemplos:
  - X is Y + 1
    > Error: Variable no instanciada en expr. Arit.
  - Y=0, X is Y + 1.
    >X = 1
  - X = 0, X is X + 1
    > No.
    No se puede unificar X con X + 1.
  - X = 0, X1 is X + 1.
    >Yes. X = 0, X1 = 1.
    Usar contadores en Prolog es un "pelín incómodo"

Uso del fallo en PROLOG.

- Hacer que un programa falle de forma deliberada
  1. El backtracking se dispara cuando algún predicado completamente instanciado falla.
  2. Hay situaciones en las que se puede forzar el backtracking de forma deliberada usando el átomo "siempre falso": `fail/0`

Variable	X
Dominio	[1..4]

Ejemplo: Recorrer un conunto de valores.
selecValor(X) es un procedimiento que instancia X con cada uno de los valores de su dominio y los muestra en pantalla.

```
valor(1).
valor(2).
valor(3).
valor(4).


selecValor(X):-
  valor(X),
  write("Valor: "),write(X),nl,
  fail.
selecValor(X):-
  write("Seleccionados todos los valores."),nl.

top:-
  selecValor(X).
```

Esquema simple para resolver PSR

- Generar y comprobar
  1. Asignar valores, dentro del dominio, para cada una de las variables
  2. Comprobar que las asignaciones verifican las restricciones
  3. Si no es así y quedan más valores, volver a 1

Variables	X,Y
Dominios	[1..4]
Restricciones	X < Y Y> 3 X <> 2

Csp0(X,Y) es un procedimiento para encontrar una asignación de valores para las variables X,Y que cumplen las Restricciones del problema

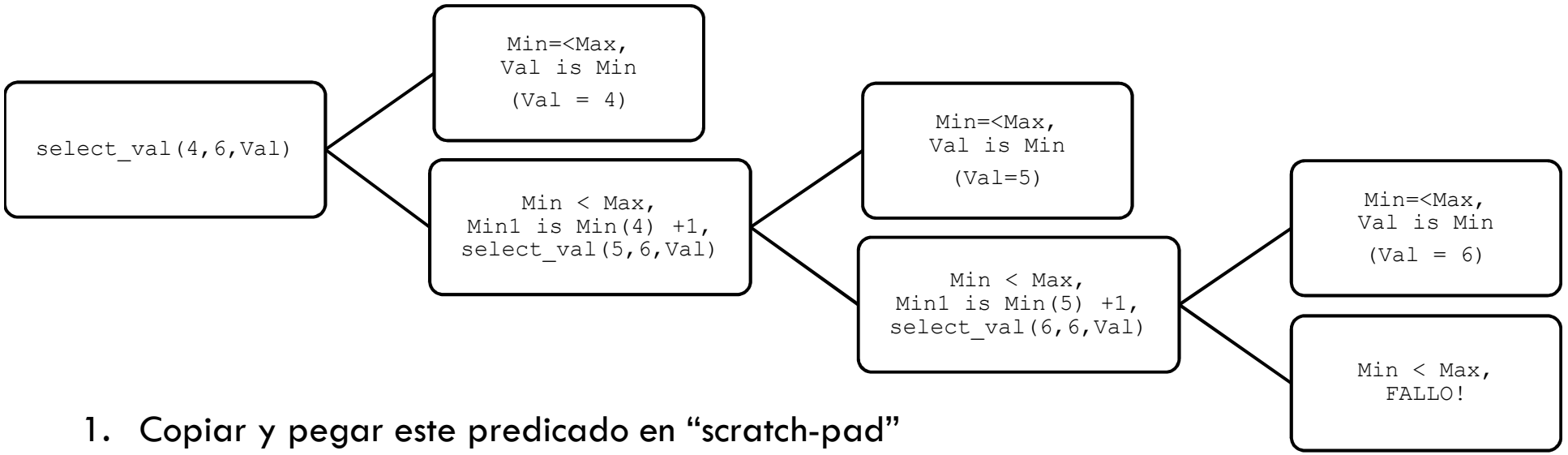
```
%Dominio de X
valorX(1).
valorX(2).
valorX(3).
valorX(4).
%Dominio de Y
valorY(1).
valorY(2).
valorY(3).
valorY(4).

cspSimple(X,Y):-
  valorX(V1), X is V1, %seleccionar valor X
  valorY(V2), Y is V2, %seleccionar valor Y
  %%Comprobación de restricciones
  X<Y,
  Y>3,
  X=\=2.
```

Selección de valores entre un rango definido

```
%select_val(Min,Max,Val). Selecciona un valor para la variable
%Val en el rango [Min,Max] ambos incluidos

select_val(Min,Max,Val) :- Min =< Max, Val is Min.
select_val(Min,Max,Val) :- Min < Max, Min1 is Min + 1,
                           select_val(Min1,Max,Val).
```



1. Copiar y pegar este predicado en “scratch-pad”
2. Llamar al predicado `select_val(4,6,Val)`
3. Observar la salida y pinchar en botón “more”.
4. ¿Cómo podemos implementar con fail el comportamiento del botón “more”?

Ejemplo Simple

PSR

Variables	X,Y
Dominios	[1..4]
Restricciones	X < Y Y> 3 X <> 2

`csp0(X,Y)` es un procedimiento para encontrar una asignación de valores para las variables X,Y que cumplen las Restricciones del problema.
Observación: cada vez que una restricción no se cumple se produce un **fallo** que dispara backtracking.

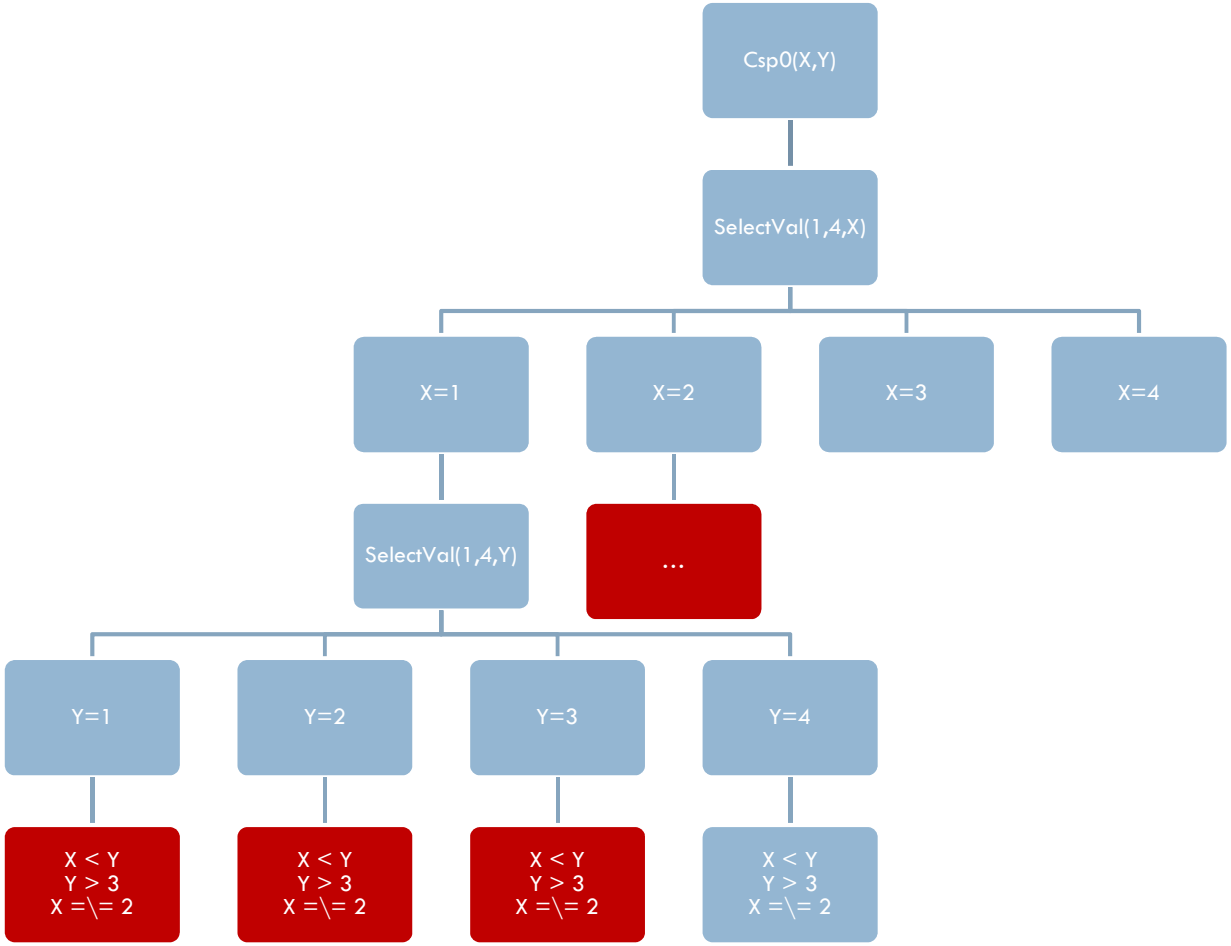
Prolog

```
csp0(X,Y) :-
   select_val(1,4,X),
   select_val(1,4,Y),
   X < Y,
   Y > 3,
   X =\= 2.

Ejecutar:
   csp0(X,Y)
   findall([X,Y],csp0(X,Y),L)
```

http://eclipseclp.org/doc/bips/kernel/allsols/findall-3.html

Árbol de Búsqueda



A continuación

- Listas en Prolog
- Modos de variables
- Generar listas
- Operaciones básicas con listas
- Solución simple a un CSP usando listas
- Solución a CSPs usando las librerías de Eclipse-Prolog

Listas en Prolog

- Lista: una secuencia de cualquier número de "items" ENTRE corchetes
  - [a, [1,2,3],tom, 1995, fecha(1,mayo,1995)]
  - Nos vamos a centrar en listas de números.
- Lista vacía: []
- Lista = [Cabeza | Cola]
  - Cabeza: primer elemento de la lista
  - Cola: Una lista formada por el resto de la lista

[X|Y] = [a,b,c]
> X=a, Y=[b,c]

  - Representan a la misma lista:
    - [a,b,c] = [a| [b,c]] = [a, b |[c]] = [a,b,c | []]

Operaciones básicas con listas

- Observar
  - El concepto de Lista es una definición recursiva y la mayoría de predicados con listas como argumentos van a ser recursivos. Los más usados son los siguientes:
- **Miembro**
  - miebro(X,L) : determina si X es miembro de la lista L.

```
miembro(X, [X|_]).
miembro(X, [ _ | Tail]) :- miembro(X,Tail).
```

  - **variable anónima:** Observar el uso del guión bajo "_":
    - Significa que no nos importa el valor con el que pueda ser instanciada la variable y no estamos interesados en conocer ese valor.

```
miembro(X, [X|_]).
miembro(X, [ _ | Tail]) :- miembro(X,Tail).

top:-
   miembro(X,[1,2,3,4]),
   writeln(X),
   fail.

top:-
   writeln(".. Y estos son Todos los valores de la lista")
```

Concatenación

- append(L1,L2,Result) Si Result es la concatenación de L1 y L2

```
append([],L,L).
append([X|L1], L2, [X|Result]) :-
                        append(L1,L2,Result).
```

- append([a,b,c], [1,2,3], L).

  > L = [a,b,c,1,2,3]

- append(L1, L2, [a,b,c]).

  > L1 = []
    L2 = [a,b,c];
  > L1 = [a]
    L2 = [b,c];
  > L1 = [a,b]
    L2 = [c];
  > L1 = [a,b,c]
    L2 = [];
  > no

- append(Before, [4|After], [1,2,3,4,5,6,7]).

  > Before = [1,2,3]
    After = [5,6,7]

- conc(_, [Pred, 4, Succ |_], [1,2,3,4,5,6,7]).

  > Pred = 3
    Succ = 5

Operaciones más usadas con listas

Library(lists)

- http://eclipseclp.org/doc/bips/lib/lists/index.html
- `member(?Term, ?List)`
- `length(?List, ?N)`
- `append(?List1, ?List2, ?AppendedList)`
- `reverse(+List, ?Reversed)`
- `sort(+List, -Sorted)`
- `flatten(+NestedList, ?FlatList)`
  - `?- flatten([[1, 2, 3], 2, [], [[3], 4], 5], L).`
  - `L = [1, 2, 3, 2, 3, 4, 5]`
  - `Yes`

Modos de variables (grounded, instantiated)

- **Modo de variable:** el rol que juega un argumento en un predicado interno de PROLOG.
  - Como ***entrada:*** tiene que estar determinada antes de llamar al predicado. Debe haber sido *ligada* a algún otro predicado, lista, átomo o número. (Fuente común de error)
  - Como ***salida***: es determinada por el predicado considerado.
  - La documentación de PROLOG distingue para cada predicado el modo de sus argumentos:
    - Modo +: ***Variable de entrada ligada e instanciada.***Una variable de entrada debe estar ligada a algún predicado,lista, átomo o número.
    - Modo ++: ***Variable de entrada ligada y "groundeada".*** Una variable de entrada debe estar ligada a algún predicado, lista, átomo o número exclusivamente con constantes.
    - Modo -: ***Variable de salida.*** No debe estar ligada a nada.
    - Modo ?:. ***Variable de entrada y salida.*** Puede estar en cualquier estado a la entrada, pero sale ligada a algo.

Modos de variables (grounded, instantiated)

- **Modo de variable:** el rol que juega un argumento en un predicado interno de PROLOG.
  - Como ***entrada:*** tiene que estar determinada antes de llamar al predicado. Debe haber sido *ligada* a algún otro predicado, lista, átomo o número. (Fuente común de error)
  - Como ***salida***: es determinada por el predicado considerado.

Variable instantiated (+X)	Variable grounded (++X)	Variable free (-X)	Variable any mode (?X)
An input bounded to any predicate or list, to an atom or number	An input bounded to a grounded predicate or list, to an atom or number	An output bounded to nothing	An input or output, bounded or free

Generar Listas

- Para hacer cualquier cosa en Prolog hay que usar listas.
- Las listas se generan habitualmente desde conjuntos de datos.
- Predicado interno findall/3
  http://eclipseclp.org/doc/bips/kernel/allsols/findall-3.html
- findall(?Term, +Goal, -List)
- donde List es la lista de todos los valores de Term para los que se satisface Goal.

```
valor(1).
valor(2).
valor(3).
escribeLista([]).
escribeLista([H|T]):-
    write(H),
    escribelista(T).
top :-
    findall(X, valor(X), L).
    escribeLista(L).
```

Ejemplo Simple Usando Listas

PSR

Variables	X,Y
Dominios	[1..4]
Restricciones	X < Y Y> 3 X <> 2

Prolog

```
csp0(X,Y) :-
   member(X,[1,2,3,4]),
   member(Y,[1,2,3,4]),
   X < Y,
   Y > 3,
   X =\= 2.

Ejecutar:
   csp0(X,Y)
   findall([X,Y],csp0(X,Y),L)
```

Estructura general para resolver PSR en Prolog

PROLOG

```
soluciona(List) :-
    declaraDominios(List),
    busqueda(List),
    comprueba(List).
```

- Inconveniente:
  - Las variables en las expresiones sobre restricciones tienen que estar instanciadas a valores para poder evaluarse
  - Por eso el proceso de búsqueda (asignación de valores a variables) necesariamente tiene que realizarse antes de la comprobación de las restricciones
  - Esto lleva a algoritmos muy ineficientes
  - Es muy laborioso
    - definir heurísticas genéricas para la selección de variables y valores.
    - Definir mecanismos de propagación de restricciones.

Estructura general para resolver PSR en ECLipSE

ECLIPSE PROLOG

```
:- lib(<librería>)
soluciona(List) :-
    declaraDominios(List),
    especificaRestricciones(List),
    busqueda(List).
```

- Establece expresividad para especificar PSR
- Sintaxis específica para declarar variables y dominios
- Expresiones para restricciones unarias, binarias, booleanas, globales, lineales, no-lineales
- Manejo de problemas con distinta representación:
  - Dominios simbólicos
  - Dominios finitos y discretos
  - Dominios intervalares
  - Dominios contínuos
- Diseño de estrategias de búsqueda
  - Configuración de algoritmos de búsqueda
  - Configuración de heurísticas
  - Mecanismos de propagación internos.

- Iteración en ECLiPsE
- Propagación de restricciones en ECLiPsE

Iteración sobre elementos de una lista

Prolog

```
write_list(List) :-
      write("Lista: "),
      write_list1(List).

write_list1([]).
write_list1([X|T]) :-
      write(X),
      write_list1(T).
```

ECLiPSe

```
write_list(List) :-
      write("Lista: "),
      ( foreach(X,List) do
            write(X)
      ).
```

Iteración sobre un rango de números

Prolog

```
write_nat(N) :-
        write("Numeros: "),
        write_nat1(0,N).

write_nat1(N, N) :- !.
write_nat1(I0, N) :-
        I is I0+1,
        write(I),
       write_nat1(I, N).
```

ECLiPSe

```
write_nat(N) :-
      write("Numbers: "),
      ( for(I,1,N) do
            write(I)
      ).
(for (I,1,N) do
   (for (J,1,N) do
      ,,,,))
(multifor ([I,J], 1,N) do
  ....
)
```

Propagación en ECLiPSe

- La propagación es parte del mecanismo de control propio,
  - es decir, en un programa ECLiPse no hay que especificar que se propage, la propagación se dispara cada vez que una variable se instancia a algún valor o se define una restricción.
- Consiste en revisar los dominios del resto de variables Y que estén relacionadas con X
- Esto es, ECLiPSe aplica nodo-consistencia y arco-consistencia (AC-3) en cada etapa de asignación de variables
  - **Nodo-consistencia**: los valores del dominio de cada variable son consistentes con las restricciones unarias en que esté implicada la variable.
  - **Arco-consistencia:** los valores del dominio de cada variable son consistentes con las restricciones binarias en que esté implicada la variable.
- Vamos a ver ahora ejemplos de programas de restricciones en los que sólo se usa propagación.

- Este mecanismo está implícito en las tres librerías que podremos usar en las prácticas:
  - sd: symbolic domains
    - `library(sd)` http://eclipseclp.org/doc/bips/lib/sd/index.html
  - fd: finite domains. Dominios finitos
    - library(fd) http://eclipseclp.org/doc/bips/lib/fd/index.html
    - library(fd_global) http://eclipseclp.org/doc/bips/lib/fd_global/index.html
  - ic: interval constraints. Dominios finitos/infinitos discretos (enteros).
    - library(ic) http://eclipseclp.org/doc/bips/lib/ic/index.html
    - library(ic_global) http://eclipseclp.org/doc/bips/lib/ic_global/index.html
    - library(ic_cumulative) http://eclipseclp.org/doc/bips/lib/ic_cumulative/index.html

Librería sd

- Dominios simbólicos http://eclipseclp.org/doc/bips/lib/sd/index.html
  - En realidad se hace un mapeo interno con enteros
- Declaración de variables y dominios
  - *Variable* **&::** Lista_de_valores_del_dominio
  - *Lista_de_variables* **&::** Lista_de_valores_del_dominio
- Restricciones: solo es posible declarar dos tipos de restricciones, igualdad y desigualdad.
  - &=
  - &\=
- **Predicados internos más usados:**
  - **get_domain_as_list(?X,-Dom)**
    - Devuelve en Dom la lista de valores del dominio de X.
  - **indomain(?X)**
    - Instancia X a un valor de su dominio
    - También en fd e ic.

Ejemplo

$$V = \{X, Y, Z\}$$
$$D(X) = \{a, c, d\}$$
$$D(Y) = \{a, b, c, d\}$$
$$D(Z) = \{c\}$$
$$C = \{X = Y, Y \neq Z\}$$

```
:-lib(sd).

csp(List):-
  List = [X,Y,Z],
  X&::[a,c,d],
  Y&::[a,b,c,d],
  Z&::[c],
  listaDominios(List,Doms):-
  (foreach(V,List),
   foreach(D,Doms) do
    get_domain_as_list(V,D)),
  X&=Y,
  Y&\= Z,
  (foreach(V,List),
   foreach(D,NewDoms) do
    get_domain_as_list(V,D)),

  (foreach(D,Doms) do
    writeln(D)),
  (foreach(D,NewDoms) do
    writeln(D)).
```

A veces, sólo con propagación se encuentra solución.

```
:-lib(sd).

csp(List):-
  List=[X,Y,Z,U],
  X&::[a,b,c],
  Y&::[a,b,d],
  Z&::[a,b],
  U&::[b],
  X&=Y,
  Y&\=Z,
  Z&\=U,
  (foreach(V,List) do
    get_domain_as_list(V,D)
    writeln(D)).
```

Coloreado de mapas

- 7 regiones en Australia
- 3 colores
- regiones vecinas con distinto color

Variables

X = {WA,NT,Q,NSW,V,SA,T}

Dominios

D_i={red,green,blue}

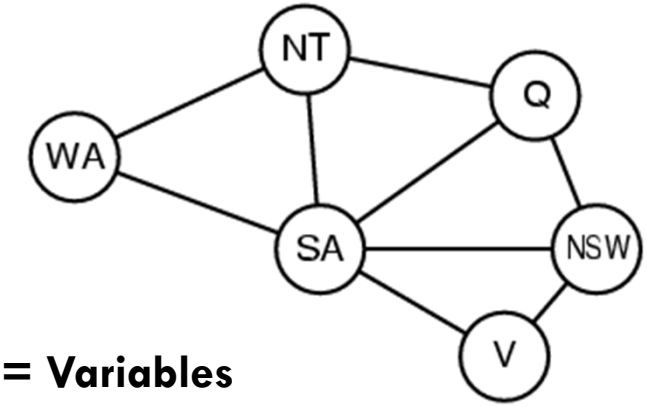
Restricciones

C = {SA ≠ WA, SA ≠ NT, SA ≠ Q,SA ≠ NSW, SA ≠ V,
WA ≠ NT, NT ≠ Q, Q ≠ NSW, NSW ≠ V}

Solución: asignación completa y consistente

{WA = red, NT = green,Q = red,NSW = green,V = red,SA = blue,T = green}

- Grafo de Restricciones



NODOS = Variables
ARCOS=Restricciones

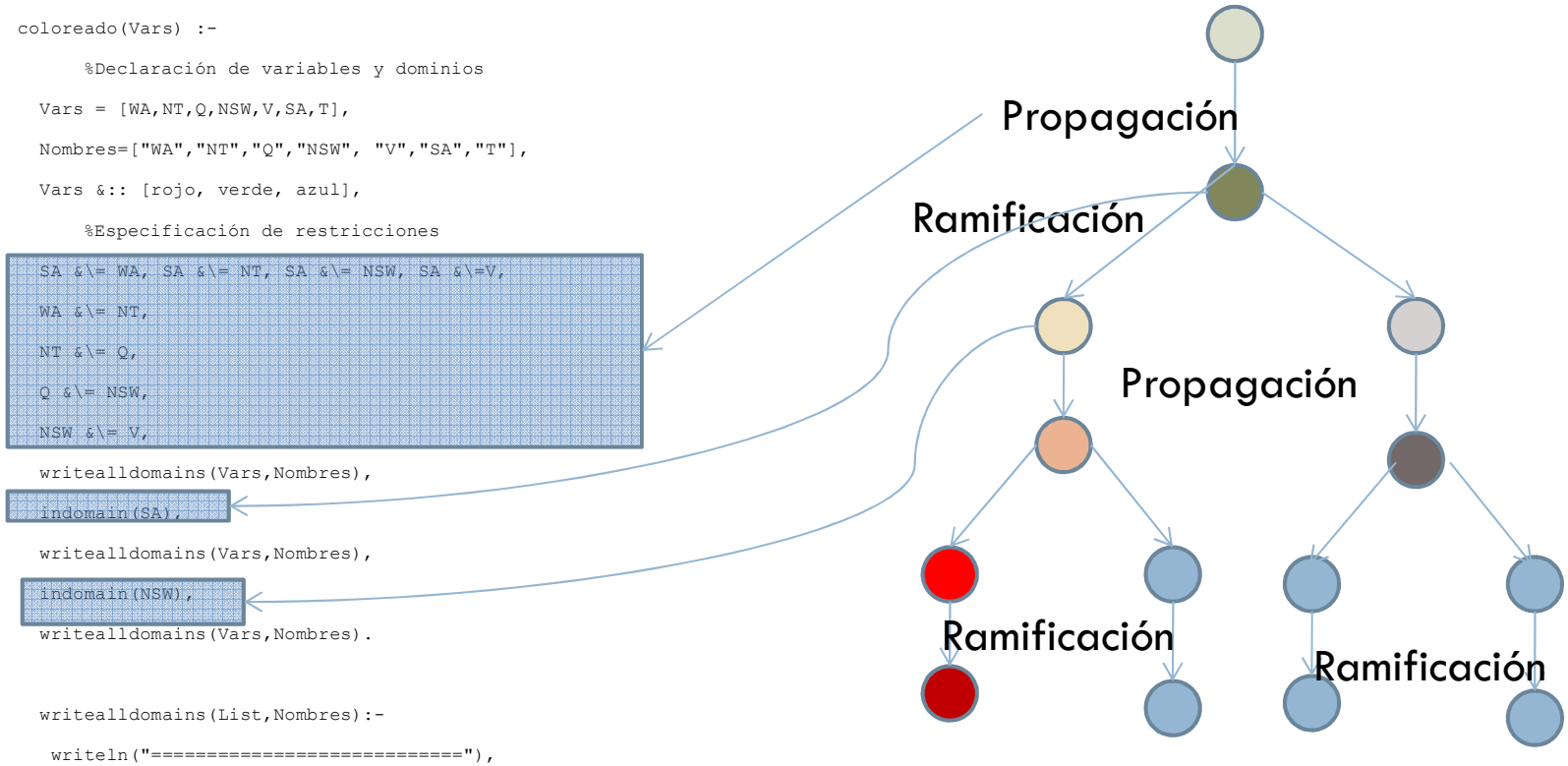
Ejemplo coloreado de mapas

```
:- lib(sd).

coloreado(Vars) :-
      %Declaración de variables y dominios
  Vars = [WA,NT,Q,NSW,V,SA,T],
  Nombres=["WA","NT","Q","NSW", "V","SA","T"],
  Vars &:: [rojo, verde, azul],
      %Especificación de restricciones
  SA &\= WA, SA &\= NT, SA &\= NSW, SA &\=V,
  WA &\= NT,
  NT &\= Q,
  Q &\= NSW,
  NSW &\= V,
  writealldomains(Vars,Nombres),
  indomain(SA),
  writealldomains(Vars,Nombres),
  indomain(NSW),
  writealldomains(Vars,Nombres).

  writealldomains(List,Nombres):-
   writeln("============================"),
   (foreach(L,List),foreach(N,Nombres) do
     write(N),write("  "),
     writedomain(L)).

  writedomain(X):-
   get_domain_as_list(X,Dom),
   writeln(Dom).
```



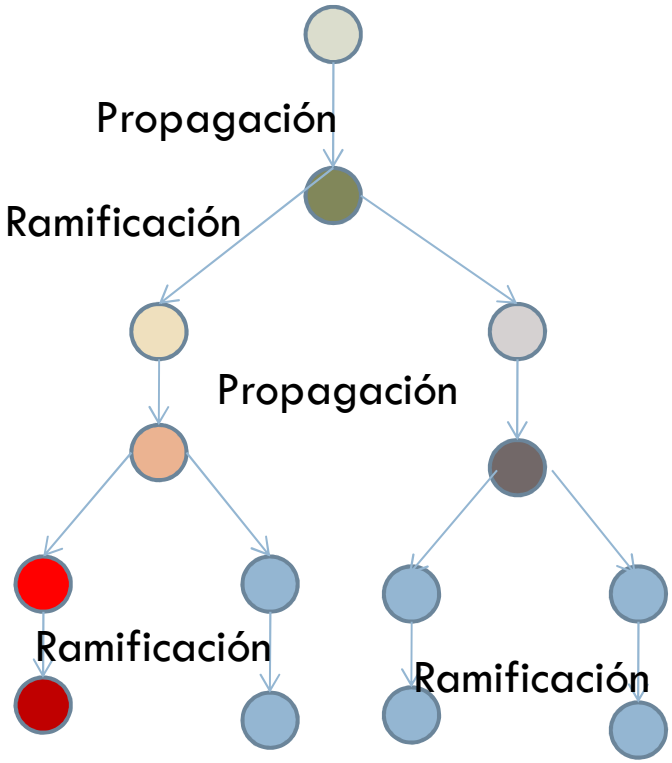
Ejemplo coloreado de mapas

```
:- lib(sd).

coloreado(Vars) :-
      %Declaración de variables y dominios
  Vars = [WA,NT,Q,NSW,V,SA,T],
  Nombres=["WA","NT","Q","NSW", "V","SA","T"],
  Vars &:: [rojo, verde, azul],
      %Especificación de restricciones
  SA &\= WA, SA &\= NT, SA &\= NSW, SA &\=V,
  WA &\= NT,
  NT &\= Q,
  Q &\= NSW,
  NSW &\= V,
  writealldomains(Vars,Nombres),
  indomain(SA),
  writealldomains(Vars,Nombres),
  indomain(NSW),
  writealldomains(Vars,Nombres).

  writealldomains(List,Nombres):-
   writeln("============================"),
   (foreach(L,List),foreach(N,Nombres) do
     write(N),write("  "),
     writedomain(L)).

  writedomain(X):-
   get_domain_as_list(X,Dom),
   writeln(Dom).
```



Ejemplo coloreado de mapas

```
:- lib(sd).

coloreado(Vars) :-
  %Declaración de variables
  Vars = [WA,NT,Q,NSW,V,SA,T],
  %Declaración de dominios
  Vars &:: [rojo, verde, azul],

  %Especificación de restricciones
  SA &\= WA, SA &\= NT, SA &\= NSW,
  SA &\=V,
  WA &\= NT,
  NT &\= Q,
  Q &\= NSW,
  NSW &\= V,

  %Búsqueda: selección/asignación de
   valores a cada una de las variables.
  (foreach(V,Vars) do
     indomain(V)).
```

- El dominio de las Variables cambia (se va estrechando) en cada llamada a indomain(X).

Librería ic

Sintaxis

- Hereda las restricciones de suspend(enteros,#, reales,$).
- Declaración de dominios
  - X::[1..13]
  - X::[3,4,5,7]
  - X::[1..3,5,7..9]
- Una variable sin especificar dominio: [-1.0inf .. 1.0inf]
- Propagación para restricciones artiméticas
  - Incluyendo operaciones escalares o entre variables
  - R. lineal: 4*X+3*Y-X<=5*Z+7-Y
  - R.artimética: 4*X+X*Y-X<=Y*(Z+5)-3*U

Ejemplo

- `[X,Y]::[1..4], X/2 - Y/2 #=1.`
- `Resultado: X=4, Y=2`
- `Excluye 3/2 y 1/2 por que no son enteros.`

Ejemplo: Puzzle Criptoaritmético

Problema

Reemplazar cada letra por un dígito en la suma:

```
  SEND
 +MORE
------
 MONEY
```

Modelo 1: sin acarreos

Variables: [S,E,N,D,M,O,R,Y]

Dominios: [0..9]

Restricciones:

Lineales

```
1000*S+100*E+10*N+D
+1000*M+100*O+10*R+E =
10000*M +1000*O+100*N+10*E+Y
```

Desigualdades:

```
S y M <> 0
```

28 Desigualdades X<>Y, para todo X,Y talque X ordenada Y

Modelo 2: con acarreos

Variables:[C1,C2,C3,C4], [S,E,N,D,M,O,R,Y]

Dominios:[0,1],[0..9]

28 Desigualdades.

Restricciones Lineales:

```
D+E=10*C1 + Y
C1+N+R=10*C2 + E
C2+E+O=10*C3 + N
C3+S+M=10*C4 + O
C4=M
```

Ejemplo: Puzzle Criptoaritmético

Problema

Reemplazar cada letra por un dígito en la suma:

```
  SEND
 +MORE
 MONEY
```

```
:-lib(ic).

sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],

  %Restricciones

    alldifferent(Digits),
    S #\= 0,
    M #\= 0,
                1000*S + 100*E + 10*N + D
              + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y,

% busqueda
    (foreach(Var,Digits) do
       indomain(Var)).
```

Modelo 1: sin acarreos

Variables: [S,E,N,D,M,O,R,Y]

Dominios: [0..9]

Restricciones:

Lineales

```
          1000*S+100*E+10*N+D
         +1000*M+100*O+10*R+E =
10000*M +1000*O+100*N+10*E+Y
```

Desigualdades:

```
          S y M <> 0
```

28 Desigualdades X<>Y, para todo X,Y talque X ordenada Y

Modelo 2: con acarreos

Variables:[C1,C2,C3,C4], [S,E,N,D,M,O,R,Y]

Dominios:[0,1],[0..9]

28 Desigualdades.

Restricciones Lineales:

```
  D+E=10*C1 + Y
C1+N+R=10*C2 + E
C2+E+O=10*C3 + N
C3+S+M=10*C4 + O
    C4=M
```

labeling(List)

- Para la lista de variables List
  - Selecciona una variable X de List
  - La instancia a un valor de su dominio (indomain(X))
- Realiza una búsqueda con backtracking sobre la lista de variables List
- Tiene en cuenta el proceso de propagación.
- Es el mecanismo estándar de búsqueda y propagación en cualquier lenguaje de resolución de PSR.

Ejemplo: Puzzle Criptoaritmético

Problema

Reemplazar cada letra por un dígito en la suma:

```
  SEND
 +MORE
------
 MONEY
```

```
:-lib(ic).


sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
  %Restricciones
    alldifferent(Digits),
    S #\= 0,
    M #\= 0,
                1000*S + 100*E + 10*N + D
              + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y,
% busqueda
    labeling(Digits).
```

Modelo 1: sin acarreos

Variables: [S,E,N,D,M,O,R,Y]

Dominios: [0..9]

Restricciones:

Lineales

```
          1000*S+100*E+10*N+D
         +1000*M+100*O+10*R+E =
10000*M +1000*O+100*N+10*E+Y
```

Desigualdades:

```
S y M <> 0
```

28 Desigualdades X<>Y, para todo X,Y talque X ordenada Y

Modelo 2: con acarreos

Variables:[C1,C2,C3,C4], [S,E,N,D,M,O,R,Y]

Dominios:[0,1],[0..9]

28 Desigualdades.

Restricciones Lineales:

$$D+E=10*C1 + Y$$

$$C1+N+R=10*C2 + E$$

$$C2+E+O=10*C3 + N$$

$$C3+S+M=10*C4 + O$$

$$C4=M$$

Restricciones globales

Restricciones globales

Restricciones que afectan a un conjunto de orden mayor que 2.

En general se aplican sobre listas de variables.

Las más comunes son accesibles desde cualquier librería.

Una vez enfrentado con un problema, es importante saber si hay alguna restricción global que puede ayudarnos a modelarlo.

alldifferent(VarList)

```
alldifferent(L):-
   (fromto(L,[X|Tail],Tail,[]) do
     (foreach(Y,Tail), param(X) do
        X$\=Y))
```

sumlist(+List,?Sum)

- List es una lista de enteros o variables
- sum(ExprList) evalúa la suma de ExprList.

•Documentación sobre restricciones globales:
  •Breve explicación sobre global constraints:
  http://eclipseclp.org/doc/tutorial/tutorial058.html
  •Librerías de restricciones globales en ECLIPSE:
  http://eclipseclp.org/doc/libman/libman021.html

Global Constraint Catalog a dictionary for Constraint Programming - current web version: 2014-06-05
http://sofdem.github.io/gccat/


XXV
DECSAI