Algoritmos para Satisfacción de Restricciones

Javier Larrosa
Dep. LSI, UPC
Jordi Girona Salgado 1-3
08034 Barcelona
larrosa@lsi.upc.es

Pedro Meseguer
IIIA-CSIC
Campus UAB
08193 Bellaterra
pedro@iiia.csic.es

Resumen

Este artículo describe los algoritmos principales que se usan en la resolución de problemas de satisfacción de restricciones incluyendo los pseudo-códigos correspondientes (a diferencia de [3] donde se hace una descripción más informal). Consideramos tres aproximaciones algorítmicas: métodos de búsqueda, de inferencia y métodos híbridos. Los métodos de búsqueda se dividen en sistemáticos y no sistemáticos. Presentamos el esquema de *backtracking* como ejemplo de búsqueda sistemática, y el de búsqueda local como ejemplo de no sistemática. Los metodos de inferencia se dividen en completos e incompletos. Describimos la consistencia adaptativa como método de completo, y varios algoritmos de consistencia local como de métodos incompletos. También describimos algunos ejemplos de métodos híbridos que combinan búsqueda con inferencia.

Palabras clave: satisfacción de restricciones, búsqueda heurística.

1 Introducción

Mientras que el artÌculo introductorio a la monografía [3] presenta una introducción general a los problemas de satisfaccion de restricciones o CSP (acrónimo del inglés *Constraint Satisfaction Problem*), este artículo trata en detalle de los algoritmos que resuelven dichos problemas. Un CSP viene definido por una *red de restricciones* (X, D, C) donde,

- $X = \{x_1, x_2, \ldots, x_n\}$ es un conjunto de n *variables*,
- $D = \{D_1, D_2, \ldots, D_n\}$ es una colección de *dominios* finitos, tal que cada variable x_i toma valores en el dominio D_i; la máxima cardinalidad de los dominios es d,
- $C = \{c_1, c_2, \ldots, c_e\}$ es un conjunto de e *restricciones*. Una restricción c_i es una relación $rel(c_i)$ sobre un subconjunto de variables $var(c_i) \subset X$, denominado su ámbito. Si $var(c_i) = \{x_{i_1}, \ldots, x_{i_k}\}$, la relación $rel(c_i)$ es un subconjunto del producto cartesiano $D_{i_1} \times \ldots \times D_{i_k}$ especificando las tuplas de valores permitidas por la restricción. La aridad de c_i es $|var(c_i)|$.

Cuando el ámbito de una restricción involucra a una o dos variables, estas se escriben como subindice de la restricción. Así, c_{ij} es la restricción entre x_i y x_j y $rel(c_{ij})$ se escribe como R_{ij}. La aridad de una red es la máxima aridad de sus restricciones. Toda red de aridad mayor que 2 se puede representar por una red equivalente binaria, mediante el paso al problema dual [31] o la transformación con variables escondidas [2].

Dada una red de restricciones se pueden realizar diversas operaciones sobre ella. Una de ellas es la resolución de la red, que consiste en encontrar una asignación a las variables de forma que todas las restricciones se satisfagan simultáneamente. Esto se conoce como problema de satisfacción de restricciones o CSP. En el caso general este problema es NP-completo, y los algoritmos que lo resuelven tienen un coste temporal exponencial en el caso peor. Dada la relevancia del problema, es importante encontrar algoritmos de resolución eficientes en el caso medio, para problemas de tamaño real.

Para resolver una red de restricciones, o de forma equivalente, solucionar el CSP definido por ella, hay diversas técnicas (ver [11] para una visión global). En este artículo presentamos tres aproximaciones:

- Búsqueda. Consiste en explorar el espacio de estados del problema hasta encontrar una solución, demostrar que no existe o agotar los recursos computacionales. La búsqueda puede ser completa si el recorrido del espacio de estados es sistemático, o incompleta si se utilizan estrategias de búsqueda local. Ciertas heurísticas pueden acelerar la búsqueda de forma significativa.
- Inferencia. Dado un problema P, consiste en deducir un problema P' equivalente, que sea más fácil de resolver. La inferencia puede ser completa si de P' se extrae la solución de forma directa, o incompleta, si es necesario complementarla con búsqueda para encontrar la solución.
- Híbrida. Combinación de las aproximaciones anteriores. Sobre un esquema de búsqueda sistemática, se incorporan métodos de inferencia incompleta o completa. En el primer caso, la inferencia incompleta se ejecuta en cada nodo visitado. En el segundo, se alternan etapas de inferencia completa con etapas de búsqueda.

2 Búsqueda

Quizás la forma más directa de resolver un CSP es mediante búsqueda en el espacio de estados del problema, explorando el conjunto de todas las configuraciones posibles. Se distinguen dos tipos de búsqueda, la *búsqueda sistemática* y la *búsqueda local*. Los métodos de búsqueda sistemática visitan todos los estados que podrían ser solución. Se trata de algoritmos *completos*, que encuentran la solución si existe, o demuestran que no hay solución. Los métodos de *búsqueda local* no garantizan que se visiten todos los estados que podrían ser solución, y un estado se puede visitar varias veces. Se trata de algoritmos *incompletos*, que pueden encontrar la solución, pero si no la encuentran no significa que no exista. Presentamos ambos tipos de búsqueda aplicados a CSP.

2.1 Búsqueda sistemática

El espacio de estados del problema se estructura mediante un árbol de búsqueda [3]. Cada nodo del árbol representa una asignaciónde variables, definida por el camino que une el nodo con la raíz. Alternativamente, cada nodo representa un subproblema definido por las variables aún no asignadas, sus dominios y las restricciones con variables no asignadas en su ámbito. Este árbol se puede recorrer de varias formas. A menudo se utiliza la búsqueda en profundidad, ya que tiene una complejidad espacial lineal y el árbol tiene profundidad acotada.

2.1.1 *Backtracking*

El algoritmo de *backtracking* [8] realiza una búsqueda en profundidad sobre el árbol del problema. En cada nodo el algoritmo comprueba si las restricciones totalmente asignadas están satisfechas. Si es así, el algoritmo prosigue la búsqueda en profundidad. Sino, la rama actual no contiene soluciones, y se realiza una vuelta atrás sobre la última variable asignada.

El código de *backtracking* para un CSP binario aparece en la Figura 1. Es la función booleana $\texttt{BT}(i, Past)$, que recibe como parámetros el índice i de la siguiente variable a asignar (variable *actual*) y el conjunto $Past$ de variables ya asignadas (variables *pasadas*; el resto de variables no asignadas, excluyendo la actual, se denominan *futuras*). $\texttt{BT}(i, Past)$ devuelve cierto si existe una solución que incluya a $Past$, en otro caso devuelve falso. Para cada valor de x_i, se comprueba que sea consistente con las asignaciones anteriores, con la función `test`. Si es así, y es la última variable, se ha encontrado una solución. Si no es la última variable, se realiza la llamada recursiva, incluyendo x_i en $Past$. Si esta llamada devuelve cierto, significa que se ha encontrado una solución por lo que devuelve cierto a su vez. En caso contrario, se continúa el bucle. Cuando se han probado todos los valores de x_i sin éxito, concluimos que no existe solución que incluya a $Past$.

Este algoritmo no especifica: (i) el orden de las variables en una rama y (ii) el orden de los valores para una variable. Estos aspectos se resuelven por medio de heurísticas de *selecciÛn de variable* y *selección de valor* [1]. Para más detalles ver [3].

Backtracking realiza una vuelta atrás cronológica: cuando detecta una inconsistencia en la rama actual, vuelve sobre la última variable asignada y cambia su valor. Otras estrategias, denominadas de *backtracking inteligente* o *look back*, permiten volver sobre una de las variables causantes de la inconsistencia, aunque no sea la última. Dichas estrategias están descritas en [3].

```
function test(i, a, Past)
  for each x_j ∈ Past do
    if (a, value(x_j)) ∉ R_ij then return false; endif
  endfor
  return true;
endfunction

function BT(i, Past)
  for each a ∈ D_i do
    x_i := a;
    if test(i, a, Past) then
      if i = n then return true;
      elsif BT(i + 1, Past ∪ {x_i}) return true;
      endif
    endif
  endfor
  return false;
endfunction
```

Figura 1. Algoritmo de *backtracking*.

2.2 Búsqueda local

Los métodos de búsqueda local intentan optimizar una función objetivo. Trabajan con asignaciones totales, que incluyen a todas las variables. Dada una asignación total inicial, el proceso itera para obtener cada vez mejores asignaciones, hasta llegar a un óptimo. Existen diversos métodos en la búsqueda local: algoritmos genéticos, *simulated annealing*, búsqueda tabú, etc. La mayoría coinciden en los elementos siguientes:

1. Función objetivo. Existe una función denominada *coste* que asigna a cada asignación total un valor numérico, en función de la bondad de dicha asignación

2. Vecindad. Dado un estado, su vecindad es el conjunto de estados a los que se puede mover en la siguiente iteración.

3. Criterio de selección. Dada una vecindad y una función de *coste*, elige el estado siguiente al actual entre su vecindad.

Un esquema genérico de búsqueda local (excluyendo los algoritmos genéticos) aparece en la Figura 2. A partir de un estado inicial A, el proceso itera mientras que que no se encuentre una solución y no se exceda el número máximo de iteraciones. En cada iteración, el criterio de selección busca un estado de la vecindad con mejor coste que el estado actual. Con esta idea, la búsqueda local puede quedar atrapada en mínimos locales, en donde ningún estado de la vecindad tenga mejor coste que el actual. Para escapar de mínimos locales, se han desallorrado estrategias que incluyen criterios aleatorios, reinicios desde otros puntos del espacio de búsqueda, permitir movimientos a estados con mayor coste con una cierta probabilidad, etc.

Considerando CSPs, la función *coste* agrega el número de restricciones no satisfechas en la asignación actual, de forma que una solución tenga coste cero. La vecindad de un estado se suele tomar como aquellos estados que difieren en los valores de una o dos variables con respecto al estado considerado. En este contexto, podemos mencionar el algoritmo de *breakout* [28], que utiliza la suma de pesos de las restricciones no satisfechas por una asignación total como *coste*. Cada vez que una restricción no se satisface, su peso se incrementa. De esta forma, la función *coste* cambia dinámicamente, permitiendo escapar de mínimos locales. También podemos mencionar el algoritmo GSAT [35], especializado para encontrar un modelo de una fórmula en lógica proposicional.

```
procedure BL(X, D, C)
  A := asig. inicial; Iter := 0;
  while Iter ≤ MaxIter ∧ ¬ sol(A) do
      Iter := Iter + 1;
      A := seleccionar(vecindad(A),coste);
  endwhile
endprocedure
```

Figura 2. Esquema de búsqueda local.

[1] Por simplicidad, todos los algoritmos del artículo se presentan con selección de variable estática lexicográfica: tras x_i la siguiente variable a asignar es x_{i+1}.

3 Inferencia

La *inferencia* en una red de restricciones consiste en deducir nuevas restricciones a partir de las restricciones existentes. Estas nuevas restricciones son redundantes con las anteriores, en el sentido de que no incorporan nuevo conocimiento.La deducción de restricciones se puede ver como el proceso de hacer explícitas ciertas restricciones que estan implícitas en la red. En el límite, la inferencia trata de sintetizar una única restricción global como conjunción de todas las restricciones explícitas de la red. Dicha restricción reemplaza a todas las restricciones explícitas, ya que captura el concimiento contenido en cada una de ellas. Obviamente, las tuplas que la satisfacen son las soluciones de la red.

El proceso de sintetizar una única restricción global se denomina de *inferencia completa*, también llamada *consistencia global*, tras el cual las soluciones se obtienen de forma directa. Otras formas de inferencia están englobadas bajo la etiqueta de *consistencia local*. Se trata de métodos de *inferencia incompleta*, capaces de hacer aflorar restricciones implícitas, pero que no sistetizan la única restricción global , y por ello no generan de forma directa las soluciones. Normalmente, la inferencia incompleta se ha de combinar con búqueda para resolver una red de restricciones.

Dos redes de restricciones son equivalentes si tienen el mismo conjunto de soluciones. El proceso de inferencia genera redes de restricciones equivalentes a la red original pero más fáciles de resolver mediante búsqueda. Por más fácil de resolver, queremos decir que el espacio de estados generado por la nueva red es más pequeño, o que se puede explorar de forma más eficiente. Por tanto, la búsqueda necesitará menos esfuerzo para encontrar una solución.

3.1 Inferencia completa

3.1.1 Operaciones con relaciones

Para tratar la inferencia en redes de restricciones, es conveniente ver una restricción como una relación R sobre un conjunto de variables, o ámbito, A. Dicha relación se denota por R_A y se compone de tuplas. Una tupla $t \in R_A$ es una secuencia finita de valores, donde cada valor, también denominado componente, corresponde a una variable de A. Definimos las siguientes operaciones sobre relaciones [11],

- Proyección. Dada una relación R_A, su proyección sobre el conjunto B, $B \subset A$, es una nueva relación $\pi_B(R_A)$ formada por las tuplas de R_A en donde se han eliminado las componentes de las variables en $A - B$. Si en este proceso aparecen tuplas duplicadas, se deja una sola copia.
- Join. Dadas dos relaciones R_A y R'_B, su join $R_A \bowtie R'_B$ es una nueva relación con ámbito $A \cup B$, definida de forma siguiente. Una tupla t pertenece a la nueva relación si y sólo si sus componentes corresponden a los de dos tuplas $r \in R_A$ y $s \in R_B$, tales que r y s tienen los mismos componentes en las variables comunes $A \cap B$.

3.1.2 Consistencia adaptativa

Sintetizar una única restricción global que capture el conocimiento contenido en todas las restricciones del problema es conceptualmente simple. Basta con calcular la restricción n-aria,

$$\bowtie_{c_i \in C} c_i$$

en donde todas las tuplas que la satisfacen son las soluciones del problema. Sin embargo, calcular esa restricción es costoso en tiempo y en espacio (exponencial en n), y es más de lo necesario para resolver el problema sin búsqueda.

Partiendo del trabajo de Freuder sobre problemas libres de backtracking [16], Dechter y Pearl han propuesto el algoritmo *ADC* (*adaptive consistency*) [12]. Este algoritmo requiere un orden estático de variables. Para cada variable, el algoritmo infiere una nueva restricción que resume el efecto de la variable en el resto del problema. Dicha variable se sustituye por esa nueva restricción, obteniendo un problema equivalente con una variable menos. Este proceso, denominado *eliminación de variables*, se repite eliminando una variable cada vez, hasta que el problema se resuelve trivialmente. Si no se ha encontrado la restricción vacía, el problema tiene solución que se construye de forma directa, sin búsqueda.

El algoritmo *ADC* aparece en la Figura 3. Se trata de la función booleana `ADC`(X, D, C), que devuelve cierto si existe solución, falso en caso contrario. Dado el orden estático de variables, estas se procesan en orden inverso, de n a 1. Si

x_i es la variable en proceso, todas las restricciones que tienen a x_i en su ámbito forman el conjunto B_i. El join de todas esas restricciones es una nueva restricción R_{A_i}. Dicha restricción se proyecta sobre el conjunto $A := A_i - \{x_i\}$, obteniendo la restricción R_A. Si R_A es la restricción vacía, el problema no tiene solución y devuelve falso. En otro caso, se elimina x_i del conjunto de variables X, se elimina B_i del conjunto de restricciones C y se añade R_A a C. Cuando se han eliminado todas las variables, la solución se obtiene de forma directa, procesando las variables de 1 a n. La variable x_i se asigna un valor compatible con las variables anteriores y con la restricción R_{A_i}. Este algoritmo tiene una complejidad espacial $O(nd^{w*})$ y temporal $O(n(2d)^{w*+1})$, donde $w*$ es la anchura inducida del grafo de restricciones con el orden estático de variables (o de forma equivalente, la aridad de la mayor restricción a construir).

function ADC(X, D, C)
  for each $i = n \dots 1$ **do**
    $B_i := \{R_{S_j} \in C \mid x_i \in S_j\}$;
    $A_i := \cup_{R_{S_j} \in B_i} S_j$;
    $R_{A_i} := (\bowtie_{R_{S_j} \in B_i} R_{S_j})$;
    $A := A_i - \{x_i\}$;
    $R_A := \pi_A(R_{A_i})$;
    if $R_A = \emptyset$ **then return** false;
    else
      $X := X - \{x_i\}$;
      $C := C \cup \{R_A\} - B_i$;
    endif
  endfor
  for each $i = 1 \dots n$ **do**
    $x_i :=$ asignacion consistente con
        previas asignaciones y R_{A_i};
  endfor
  return true;
endfunction

Figura 3. Algoritmo de consistencia adaptativa.

3.2 Inferencia incompleta

Sintetizar la restricción global de una red de restricciones puede parecer complicado. Un problema inicialmente más asequible es el de resolver subredes de un tamaño prefijado, por ejemplo, subredes de una variable, subredes de dos variables, etc.

Formalmente, una subred (X', D', C') de una red de restricciones (X, D, C), esta definida por un subconjunto de variables $X' = \{x_1, \dots, x_k\}$, $X' \subset X$, sobre los dominios originales $D' = (D_1, \dots, D_k)$, bajo el conjunto de restricciones $C' = \{c \mid c \in C \wedge var(c) \subseteq X'\}$. Encontrar la solución de una subred no garantiza que el problema tenga solución. Sin embargo, los resultados negativos encontrados en este proceso sí se pueden aplicar a la red inicial. Si resolviendo una subred encontramos que ciertos valores o combinaciones de valores no aparecen en ninguna solución de la subred, estos tampoco aparecerán en las soluciones de la red. La justificación es directa: todas las restricciones de la subred estan presentes en la red inicial, $C' \subset C$, luego todos los valores o combinaciones prohibidos por la subred también lo serán por la red inicial. Si la subred no tiene solución, la red no tiene solución.

Con este proceso, denominado de consistencia local, no encontramos la solución del problema, pero deducimos nuevas restricciones que se añaden al problema, y permiten acelerar la búsqueda de la solución global. Si se deduce la restricción vacía, el problema no tiene solución.

A continuación, presentamos métodos de inferencia incompleta para redes de restricciones binarias. Consideramos subredes de uno, dos y tres nodos, [24], y su generalización a subredes de k nodos.

3.2.1 Consistencia de nodos

Una variable x_i es *nodo consistente* ssi todo valor de su dominio D_i está permitido por las restricciones unitarias sobre x_i. Una red de restricciones es nodo consistente ssi toda variable es nodo consistente. Una red se convierte en nodo consistente con el procedimiento NC-1 que aparece en la Figura 4, y es equivalente a la operación,

$$D_i \leftarrow D_i \cap R_i \qquad i = 1, \dots, n$$

procedure NC-1(X, D, C)
  for each $x_i \in X$ **do**
    for each $a \in D_i$ **do**
      if $a \notin R_i$ **then** $D_i := D_i - \{a\}$; **endif**
    endfor
  endfor
endprocedure

Figura 4. Procedimiento NC-1.

3.2.2 Consistencia de arcos

Un valor $a \in D_i$ es arco consistente con respecto a la variable x_j ssi existe otro valor $b \in D_j$ tal que la tupla $(a,b) \in R_{ij}$. Una variable x_i es arco consistente con respecto a x_j ssi todos los valores de su dominio lo son. Una variable x_i es arco consistente si lo es con respecto a cualquier variable. Una red de restricciones es arco consistente ssi lo son todas sus variables.

Una variable x_i se hace arco consistence con respecto a la variable x_j por medio de la función `revise`(i,j) (Figura 5), borrando de D_i aquellos valores que no son compatibles con ningún valor de x_j. El efecto de la función `revise`(i,j) es equivalente a la operación,

$$D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$$

Una red de restricciones binaria se convierte en arco consistente mediante el procedimiento `AC-3` [24] que aparece en la Figura 5, mediante la reiterada aplicación de la función `revise`(i,j). Inicialmente, la variable Q contiene todas las restricciones del problema en ambos sentidos. Si al hacer arco consistente la restricción c_{ij} de i a j (se trata de un proceso direccional) se modifica D_i, se vuelven a introducir en Q todas aquellas restricciones que se pudieran haber vuelto arco inconsistentes por este proceso. El algoritmo `AC-3` tiene una complejidad temporal $O(ed^3)$.

function `revise`(i,j)
  cambio:= false;
  for each $a \in D_i$ **do**
    if $\neg\exists b \in D_j$ tal que $(a,b) \in R_{ij}$ **then**
      $D_i := D_i - \{a\}$;
      cambio := true;
    endif
  endfor
  return cambio;
endfunction

procedure `AC-3`(X,D,C)
  $Q := \{(i,j)(j,i) \mid c_{ij} \in C\}$;
  while $Q \neq \emptyset$ **do**
    extrae (i,j) de Q;
    if `revise`(i,j) **then**
      $Q := Q \cup \{(k,i) \mid c_{ki} \in C, k \neq j, k \neq i\}$;
    endif
  endwhile
endprocedure

Figura 5. Procedimiento `AC-3` y función `revise`.

Sustituyendo en `AC-3` la función `revise` por la función `revise2001`, obtenemos el procedimiento `AC-2001` [7]. La función `revise2001` aparece en la Figura 6, y presenta dos novedades con respecto a `revise`. Primero, existe un orden total $>_d$ en cada dominio, y los valores son comprobados en ese orden. Segundo, cada vez que se revisa la restricción de x_i a x_j, para cada valor $a \in D_i$, se guarda en `last`(x_i,a,x_j) el primer valor compatible en D_j siguiendo el orden. Cada vez que se ejecuta `revise2001`(i,j), para cada valor $a \in D_i$ se comprueba si el valor `last`(x_i,a,x_j), calculado en la anterior ejecución de `revise2001`(i,j), está todavía en D_j. Si es así, a es arco consistente y no hay que hacer nada más. Sino, hay que buscar un valor $b \in D_j$ compatible con a, y que estará tras `last`(x_i,a,x_j) en el orden. Si existe, a es arco consistente y se actualiza `last`$(x_i,a,x_j) := b$. Sino, a es arco inconsistente y se ha de eliminar de D_i. Es fácil ver que la complejidad de `AC-2001` es $O(ed^2)$, que es óptima [27].

function `revise2001`(i,j)
  cambio:= false;
  for each $a \in D_i$ **do**
    if `last`$(x_i,a,x_j) \notin D_j$ **then**
      if $\exists b \in D_j,\ b >_d$ `last`$(x_i,a,x_j)\ \wedge\ (a,b) \in R_{ij}$ **then**
        `last`$(x_i,a,x_j) := b$;
      else
        $D_i := D_i - \{a\}$;
        cambio := true;
      endif
    endif
  endfor
  return cambio;
endfunction

Figura 6. Función `revise2001`.

3.2.3 Consistencia de caminos

Un par de valores $(a,b) \in R_{ij}$ de las variables x_i y x_j es camino consistente con respecto a una tercera variable x_k ssi existe $c \in D_k$ tal que $(a,c) \in R_{ik}$ y $(b,c) \in R_{jk}$. Un par de variables x_i y x_j es camino consistente con respecto a x_k ssi todas sus parejas compatibles de valores lo son. Un par de variables x_i y x_j es camino consistente ssi lo son con respecto a cualquier variable. Una red es camino consistente ssi todos los pares de variables son camino consistentes.

Un par de variables x_i, x_j se hace camino consistente con respecto a x_k por medio de la función `revise3`(i, j, k), borrando de R_{ij} aquellos pares de valores que no se pueden extender a ningún valor de x_k. El efecto de la función `revise3`(i, j, k) es equivalente a la operación,

$$R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{ik} \bowtie D_k \bowtie R_{kj})$$

Una red de restricciones binaria se convierte en camino consistente mediante el procedimiento `PC-2` [24] (Figura 7), por medio de la reiterada aplicación de la función `revise3`(i, j, k). La variable Q se inicializa con todos los posibles triángulos (i, j, k) de variables en la red. Si al hacer camino consistente la restricción c_{ij} se modifica R_{ij}, se vuelven a introducir en Q todas los triangulos que incluyen i, j y cualquier otra variable, ya que han podido dejar de ser camino consistentes. La complejidad de `PC-2` es $O(n^3 d^5)$. Un procedimiento más sofisticado es `PC-3` [27].

```
function revise3(i, j, k)
  cambio:= false;
  for each (a, b) ∈ R_ij do
    if ¬∃c ∈ D_k tal que (a, c) ∈ R_ik y (b, c) ∈ R_jk then
      R_ij := R_ij − {(a, b)};
      cambio := true;
    endif
  endfor
  return cambio;
endfunction

procedure PC-2(X, D, C)
  Q := {(i, j, k) | 1 ≤ i < j ≤ n, 1 ≤ k ≤ n, k ≠ i, k ≠ j};
  while Q ≠ ∅ do
    extrae (i, j, k) de Q;
    if revise3(i, j, k) then
      Q := Q ∪ {(l, i, j)(l, j, i) | 1 ≤ l ≤ n, l ≠ j, l ≠ i};
    endif
  endwhile
endprocedure
```

Figura 7. Proc. `PC-2` y función `revise3`.

3.2.4 k-consistencia

La consistencia de nodos elimina aquellos valores que no son posibles para una variable; la consistencia de arcos elimina aquellos valores de una variable que no se pueden extender de forma consistente a una segunda variable; la consistencia de caminos elimina aquellos pares de valores consistentes que no se pueden extender de forma consistente a una tercera variable. Siguiendo esta secuencia, podemos considerar conjuntos de $k-1$ variables y ver si se pueden extender de forma consistente a cualquier k-ésima variable. Esta idea es la base de la k-consistencia. Una red de restricciones es *k-consistente* ssi para todo conjunto de $k-1$ variables asignadas de forma consistente y cualquier k-ésima variable, existe un valor de esta variable consistente con los valores de las $k-1$ variables anteriores. Los algoritmos que alcanzan la k-consistencia [9] tienen complejidad exponencial en k, y descartan combinaciones de $k-1$ valores que no se pueden extender de forma consistente a una k-ésima variable.

4 Búsqueda + Inferencia

Las estratégias anteriores, búsqueda e inferencia, se pueden combinar dando lugar a algoritmos híbridos. Exploramos dos combinaciones:

1. Búsqueda sistemática + inferencia incompleta. Dentro de un esquema de búsqueda sistemática, se realiza un proceso de consistencia local sobre el subproblema representado en cada nodo. El proceso de consistencia local evalua la consistencia del subproblema hasta un cierto nivel, descubriendo tuplas de valores que no pueden estar en la solución. Como consecuencia, el espacio de estados del subproblema disminuye, aumentando la eficiencia de la búsqueda. Si se descubre un dominio vacío, el subproblema no tiene solución.

2. Búsqueda sistemática + inferencia completa. La inferencia completa por eliminación de variables tiene un coste espacial exponencial, dependiendo de un parámetro del grafo de restricciones. Por este motivo no es posible utilizarla en muchos problemas en los que este parámetro no está acotado (problemas con grafos de restricciones altamente cíclicos). Sin embargo sí que es posible integrar la eliminación de variable dentro de un esquema de búsqueda y aplicarla de manera limitada cuando las condiciones para la eliminación sean suficientemente propicias.

A continuación, presentamos los algoritmos *forward checking* y *MAC* como ejemplos del primer tipo, y el algoritmo *VES* (*variable elimination search*) como ejemplo del segundo.

4.1 Forward Checking

Forward checking [19] sigue una búsqueda en profundidad, realizando en cada nodo un proceso de arco consistencia sobre las restricciones parcialmente asignadas. En el caso binario, tras asignar una variable, se eliminan de los dominios futuros los valores incompatibles con el recién asignado.

El algoritmo de *forward checking* para problemas binarios aparece en la Figura 8. Se trata de la función booleana $\mathtt{FC}(i, Past, D = [D_i, \ldots, D_n])$, que recibe como parámetros el índice i de la variable actual, el conjunto $Past$ de variables pasadas y el vector $D = [D_i, \ldots, D_n]$ de dominios del resto de variables. Esta función devuelve cierto si existe una solución que incluya a $Past$, en otro caso devuelve falso. La función va probando los valores posibles de x_i. Dado un valor, si x_i es la última variable, hemos encontrado la solución. Si no es la última variable, la función $\mathtt{AC}(\{x_i, \ldots, x_n\}, [\{a\}, D_{i+1}, \ldots, D_n], C_{cf})$ realiza un proceso de arco consistencia sobre el subconjunto de restricciones $C_{cf} = \{c_{ij} | c_{ij} \in C, i < j\}$, formado por aquellas restricciones que conectan la variable actual con las futuras. Como resultado, se obtienen los nuevos dominios futuros $NewD$, eliminando aquellos valores arco inconsistentes con el recién asignado. Si no hay un dominio futuro vacío, se realiza la llamada recursiva, incluyendo x_i en $Past$ con los nuevos dominios $NewD$. Si esta llamada devuelve cierto, se ha encontrado una solución por lo que devuelve cierto a su vez. En caso contrario, se continúa el bucle. Si se han probado todos los valores de x_i sin éxito, no existe solución que incluya a $Past$.

function $\mathtt{FC}(i, Past, D = [D_i, \ldots, D_n])$
  for each $a \in D_i$ **do**
    $x_i := a$;
    if $i = n$ **then return** true;
    else
      $C_{cf} = \{c_{ij} \mid c_{ij} \in C, i < j\}$;
      $NewD := \mathtt{AC}(\{x_i, \ldots, x_n\}, [\{a\}, D_{i+1}, \ldots, D_n], C_{cf})$;
      if $NewD$ no contiene $\emptyset$ **then**
        if $\mathtt{FC}(i+1, Past \cup \{x_i\}, NewD)$ **then**
          return true;
        endif
      endif
    endif
  endfor
  return false;
endfunction

Figura 8. Algoritmo de *forward checking*.

4.2 MAC: Manteniendo arco consistencia

El algoritmo *MAC* (acrónimo de *maintaining arc consistency*) [32], mantiene la arco consistencia en los subproblemas que genera. Cada nuevo subproblema es hecho arco consistente considerando todas sus restricciones.

En la Figura 9 se presenta el algoritmo *MAC*. Se trata de la función booleana $\mathtt{MAC}(i, D)$, que recibe como parámetros el índice i de la variable actual y el vector $D = [D_1, \ldots, D_n]$ de todos los dominios del problema. Esta función devuelve cierto si existe una solución entre los valores de D, en otro caso devuelve falso. En este algoritmo, asignar una variable es equivalente a reducir su dominio a un solo valor. Inicialmente, se copian los dominios futuros recibidos a D'_j. La función va probando los valores posibles de x_i. Para cada valor a, el algoritmo considera dos subproblemas, definidos por el dominio D'_i de la variable actual:

1. $D'_i := \{a\}$ (x_i toma el valor a). Si x_i es la última variable, hemos encontrado una solución. Si no es la última variable, la función $\mathtt{AC}(X, D, C)$ realiza un proceso de arco consistencia sobre el conjunto C de todas las restricciones del problema, considerando $D = [D_1, \ldots, D_{i-1}, D'_i, \ldots, D'_n]$, los dominios actuales de las variables. Como resultado, se obtienen los nuevos dominios futuros $NewD$, eliminando aquellos valores arco inconsistentes. Si no hay un dominio futuro vacío, se realiza la llamada recursiva, con los nuevos dominios $NewD$. Si esta llamada devuelve cierto, significa que se ha encontrado una solución por lo que devuelve cierto a su vez. Sino, se continúa con la segunda opción.

2. $D'_i := D_i - \{a\}$ (x_i no puede tomar el valor a). Tanto si existe un dominio vacío en $NewD$ como si la llamada recursiva retorna falso, a ya no es un valor posible para x_i. Se realiza arco consistencia $\mathtt{AC}(X, D, C)$ sobre el nuevo dominio de x_i. Como resultado, $\mathtt{AC}$ calcula los nuevos dominios futuros $NewD$, eliminando aquellos valores arco inconsistentes. Si no hay un dominio futuro vacío, se copian los nuevos dominios futuros en D'_j y se continúa el bucle. En caso contrario, hay que salir del bucle, no hay solución en la rama actual que incluya un valor de D'_i.

Cuando se sale del bucle, se han probado todas las opciones para x_i sin éxito, por lo que no hay solución en la rama actual.

```
function MAC(i, D = [D_1, ..., D_n])
  for each j = i+1 ... n do D'_j := D_j; endfor
  for each a ∈ D_i do
    D'_i := {a};
    if i = n then return true;
    else
      NewD := AC(X, [D_1, ..., D_{i-1}, D'_i, ..., D'_n], C);
      if NewD no contiene ∅ then
        if MAC(i+1, NewD) then return true; endif
      endif
      D_i := D_i − {a};
      D'_i := D_i;
      NewD := AC(X, [D_1, ..., D_{i-1}, D'_i, ..., D'_n], C);
      if NewD contiene ∅ then salir bucle;
      else
        for each j = i+1 ... n do
          D'_j := NewD[j];
        endfor
      endif
    endif
  endfor
  return false;
```

Figura 9. Algoritmo *MAC*.

4.3 Búsqueda y eliminación de variables

Como ejemplo de combinación de búsqueda completa e inferencia completa está el algoritmo *VES* (*variable elimination search*) [23]. Eliminar una variable, que es el paso básico de la inferencia completa, tiene un coste espacial y temporal exponencial en la anchura de la variable (número de variables restringidas con ella y anteriores en el orden). Cuando la anchura es pequeña, este proceso se puede realizar, pero si la anchura es grande, el coste es prohibitivo. Por otra parte, asignar una variable modifica la topología del grafo, una vez propagado el efecto de la asignación, por lo que se puede utilizar para disminuir la anchura de ciertas variables.

El algoritmo *VES* combina estas ideas, usando un orden estatico de variables. Este algoritmo aparece en la Figura 10. Se trata de la función booleana VES(S, k, P, F, E, D, C), que recibe como parámetros el procedimiento S de búsqueda, el parámetro k, las variables pasadas P, futuras F y eliminadas E, los dominios actuales D y las restricciones actuales C. Esta función devuelve cierto si existe una solución, en otro caso devuelve falso. Si el conjunto F es vacío, se ha encontrado una solución. Sino, se escoje x_i la última variable de F y se calcula su anchura. Si esta es menor o igual que k, se elimina la variable con la función `varelim`. E Sino, se realiza búsqueda sobre los valores de la variable con la función `varbranch`. Cuando se realizan con éxito, ambas funciones llaman a la función `VES` con el subproblema que queda.

```
function VES(S, k, P, F, E, D, C)
  if F = ∅ then return true;
  else
    x_i := ultima(F);
    B_i := {R_{S_j} ∈ C | x_i ∈ S_j};
    if |B_i| ≤ k then
      return varelim(S, k, x_i, P, F, E, C);
    else
      return varbranch(S, k, x_i, P, F, E, C);
endfunction

function varelim(S, k, x_i, P, F, E, D, C)
  A := ∪_{R_{S_j} ∈ B_i} S_j − {x_i};
  R_A := π_A(⋈_{R_{S_j} ∈ B_i} R_{S_j});
  if R_A = ∅ then return false;
  else
    return VES(S, k, P, F − {x_i}, E ∪ {x_i}, C ∪ {R_A});
endfunction

function varbranch(S, k, x_i, P, F, E, D, C)
  for each a ∈ D_i do
    NewD := lookahead(S, x_i, a, P, F, C);
    if NewD no contiene ∅ then
      return VES(S, k, P ∪ {x_i}, F − {x_i}, E, NewD, C);
    endif
  endfor
  return false;
endfunction
```

Figura 10. Algoritmo *VES*.

5 Temas relacionados

Existen una serie de temas muy relacionados con los esquemas algorítmicos anteriores, que por razones de espacio no podemos presentar en detalle. A continuación, mencionamos algunos de estos temas, dando las referencias básicas que ha de seguir el lector si quiere profundizar en ellos.

5.1 Explotación de simetrías

En muchos problemas aparecen simetrías, como transformaciones que dejan invariante el conjunto de restricciones. Al aplicar una simetría sobre un estado s, se obtiene un nuevo estado s' que es equivalente a s. Una simetría induce una relación de equivalencia en el espacio de estados, agrupando en clases a todos los estados equivalentes por la simetría. Cada clase contiene soluciones (todos los estados de la clase son solución) o no soluciones (ningún estado de la clase es solución). Para resolver un CSP con simetrías, no es necesario visitar todos los estados, sino solamente un estado por clase de equivalencia. De esta forma disminuye drásticamente el tamaño del espacio y aumenta la eficiencia de la búsqueda.

Se han desarrollado diversas estrategias para explotar las simetrías en la resolución de CSP. Citamos las siguientes,

1. Nuevas restricciones para romper simetrías (*symmetry breaking constraints* [29]). Se modifica el problema original, añadiendo nuevas restricciones que rompen sus simetrías. El nuevo problema mantiene todas las soluciones no simétrias pero elimina las simétricas. Las nuevas restricciones se pueden añadir antes de comenzar la búsqueda, o durante la misma.

2. Modificación de la estrategia de búsqueda. (*symmetry breaking search* [14]). Se modifica la estrategia de búsqueda, para evitar visitar estados simétricos a los ya visitados. Para ello, se definen las soluciones canónicas, y se podan los subarboles que no las contienen.

3. Ruptura de simetrías durante la búsqueda (*symmetry breaking during search* [18]). Cuando se visita un subárbol, se añaden restricciones que evitan visitar un subárbol simétrico en el futuro.

4. Heurística de ruptura de simetrías (*symmetry breaking heuristic* [26]). Al asignar una variable se rompen simetrías. Esta heurística busca a la variable que más simetrías puede romper al ser asignada, orientando la búsqueda hacia subespacios con menor número de estados simétricos.

5.2 CSP no binarios

Durante un cierto tiempo, la investigación se centró sobre restricciones binarias, y la mayor parte de los algoritmos se desarrolló para problemas binarios. En teoría, esto no era un inconveniente para aplicar dichos algoritmos a problemas no binarios, ya que todo problema no binario se puede transformar en otro binario equivalente. En la práctica, sin embargo, esta transformación presenta diversos inconvenientes, por lo que se han desarrollado versiones no binarias de la mayoría de los algoritmos.

La generalización no binaria de ciertos algoritmos como *backtracking* es directa. El popular algoritmo de *forward checking*, admite diversas generalizaciones no binarias [5]. La versión no binaria de consistencia de arcos se denomina consistencia de arcos consistencia generalizada [6], y permite que la generalización del algoritmo *MAC* al caso no binario sea directa. Una restricción no binaria es descomponible, si se puede reescribir como un conjunto de restricciones binarias sobre las mismas variables manteniendo su semáantica. La potencia de las consistencias locales sobre restricciones n-arias y su descomposición binarias se ha comparado en [21]. Para más detalles, ver [33].

5.3 Restricciones globales

Desde el punto de vista de los algoritmos, es habitual ver las restricciones como relaciones entre variables, sin asumir ninguna semántica. Esta idea no es adecuada cuando se trabaja sobre problemas reales. Existe un conjunto de restricciones con un significado bien definido que aparecen a menudo en las aplicaciones. Explotar su semántica permite obtener procedimientos de propagación más eficientes, con lo que disminuye la complejidad de los algoritmos.

Estas restricciones suelen ser no binarias, y se denominan genéricamente *restricciones globales*. El ejemplo clásico de restricción global descomponible es la restricción *all-different*$(x_1, x_2, \dots, x_i)$ [30]. Esta restricción es equivalente a un conjunto de $\frac{i(i-1)}{2}$ restricciones binarias $\neq$ entre las variables. Si cada variable tiene $i-1$ valores, la arco consistencia sobre las restricciones binarias $\neq$ no es capaz de deducir la poda de ningun valor. Sin embargo, la arco consistencia generalizada sobre la restricción global *all-different* permite deducir que el problema no tiene solución. Por contra, la restricción global *sum* $\sum_{i \in 1 \ldots n} x_i = y$ no es descomponible, ya que no se puede expresar en restricciones de menor aridad sin introducir nuevas variables. El catálogo de restricciones globales crece a medida que se atacan nuevos problemas [4].

5.4 Entornos de programación con restricciones

Buena parte de las técnicas para resolver redes de restricciones se han encapsulado en los entornos de programación con restricciones. Dichos entornos proporcionan un estilo de programación declarativo, en donde el usuario formula su problema en términos de variables, dominios y restricciones, y pide a un resolvedor genérico que calcule la solución [34]. Estos entornos incluyen un catálogo de restricciones globales con procedimientos de propagación específicos, y heurísticas genéricas.

La programación con restricciones encaja de forma natural con la programación lógica: no hay más que reescribir las restricciones como predicados y tomar como objetivo la conjunción de dichos predicados. Sustutuyendo la unificación por satisfacción de restricciones, se obtiene la *programación lógica con restricciones* o CLP (acrónimo del inglés *Constraint Logic Programming*) [25]. Se han desarrollado diversos entornos CLP, como CHIP [20] y ECLIPSE [22]. También se han desarrollado entornos de programación con restricciones sobre lenguajes de programación imperativos (C++), como bibliotecas de procedimientos especializados que se incluyen en el programa del usuario. Como ejemplo, podemos citar el ILOG Solver [13]. Varios de estos entornos han sido comparados en [15].

Desde los primeros trabajos de Waltz en los años 70 [37], se ha avanzado mucho en la resolución de problemas de restricciones. Se han editado varios libros que contienen la teoría y los algoritmos necesarios para resolver este tipo de problemas [20, 36, 25, 17, 1, 11]. Sobre problemas específicos, existe un servidor WEB [10], que contiene descripciones de diversos problemas, así como estrategias de modelización y resolución.

Referencias

[1] K. Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[2] F. Bacchus and P. van Beek. On the conversion between non-binary and binary costraint satisfaction problems. In *Proc. AAAI*, pages 311–318, 1998.

[3] F. Barber and M.A. Salido. La programación de restricciones. una introducción. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 20, 2003.

[4] N. Beldiceanu. Tutorial on global constraints. Technical report, http://www.sics.se/isl/csp/, 2002.

[5] C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141:205–224, 2002.

[6] C. Bessière and J. C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proc. IJCAI*, pages 398–404, 1997.

[7] C. Bessière and J. C. Régin. Refining the basic constraint propagation algorithm. In *Proc. IJCAI*, pages 309–315, 2001.

[8] J.J. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18:651–656, 1975.

[9] M. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1990.

[10] CSPLIB. A problem library for constraints. maintained by I. Gent, T. Walsh, B. Selman, http://4c.ucc.ie/ tw/csplib/.

[11] R. Dechter. *Constraint Processing.* Morgan Kauffman, 2003.

[12] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[13] ILOG ed. *ILOG Solver 5.0, reference manual.* 2000.

[14] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *Proc. CP*, pages 93–107, 2001.

[15] A. J. Fernandez and P. M. Hill. A comparative study of eight constraint programming languages over the boolean and finite domains. *Constraints*, 5:275–302, 2000.

[16] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29:24–32, 1987.

[17] T. Fruhwirth and S. Abdennadher. *Essentials of constraint programming.* Springer, 2003.

[18] I. Gent and B. Smith. Symmetry breaking in constraint programming. In *Proc ECAI*, pages 599–603, 2000.

[19] M. Haralick and G. L. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[20] P. Van Hentenryck. *Constraint satisfaction in logic programming.* MIT Press, 1989.

[21] T. Walsh I. Gent, K. Sterigou. Descomposable constraints. *Artificial Intelligence*, 123:133–156, 2000.

[22] IC-PARK. The eclipse 4.2 constraint logic programming system. Technical report, http://www.icpark.ic.ac.uk/eclipse/, 1999.

[23] J. Larrosa. Boosting search with variable elimination. In *Proc. CP*, pages 291–305, 2000.

[24] A. K. Mackworth. Consistency in network of relations. *Artificial Intelligence*, 8:99–118, 1977.

[25] K. Marriot and P. J. Stuckey. *Programming with constraints.* MIT Press, 1998.

[26] P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129:133–163, 2001.

[27] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[28] P. Morris. The breakout method for escaping from local minima. In *Proc. AAAI*, pages 40–45, 1993.

[29] J. F. Puget. On the satisfiability of symmetrical constraint satisfaction problems. In *Proc. ISMIS*, pages 350–361, 1993.

[30] J. C. Régin. A filtering algorithm for constraints of difference in csps. In *Proc. AAAI*, pages 362–367, 1994.

[31] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proc. ECAI*, pages 550–556, 1990.

[32] D. Sabin and E. C. Freuder. Contradicting convenctional wisdom in constraint satisfaction. In *Proc. ECAI*, pages 125–129, 1994.

[33] M.A. Salido. Técnicas para el Manejo de CSPs no Binarios. *Inteligencia Artificial: Revista Iberoamericana de Inteligencia Artificial*, 20:95–110, 2003.

[34] C. Schulte. *Programming constraint services.* LNAI 2302, Springer, 2002.

[35] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. AAAI*, pages 440–446, 1992.

[36] E. Tsang. *Fundations of constraint satisfaction.* Academic Press, 1993.

[37] D. Waltz. Understanding line drawings of scene with shadows. In *The psychology of computer vision*, 1975.