**Solution Techniques for Constraint Satisfaction Problems**

Felip Manyá (1), Carla Gomes (2)

(1) Departamento de Informática e Ingeniería Industrial
Universitat de Lleida
Jaume II, 69, E-25001 Lleida, Spain

(2) Department of Computer Science
Cornell University
Ithaca, NY 14853 USA

e-mail: felip@eup.udl.es, gomes@cs.cornell.edu

A wide range of problems in Artificial Intelligence can be represented as instances of the Constraint Satisfaction Problem (CSP), and then be solved using some of the existing techniques for solving CSPs. In this paper, we start by defining the concept of CSP and showing how some combinatorial problems can be modelled as CSPs. Next, we give a detailed description of the basic techniques for constraint satisfaction: constraint propagation algorithms (node consistency, arc consistency, and $k$-consistency), search algorithms (generate and test, backtracking, backjumping, and conflict-directed backjumping), and hybrid algorithms (forward checking, and maintaining arc consistency).

*Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No.19 (2003),pp. 169-180*
*ISSN: 1137-3601. © AEPIA (http://www.aepia.org/revista).*

# Técnicas de resolución de problemas de satisfacción de restricciones

Felip Manyà[*]
Dpto. de Informática e Ing. Industrial
Universitat de Lleida
Jaume II, 69, E-25001 Lleida, España
felip@eup.udl.es

Carla Gomes
Dept. of Computer Science
Cornell University
Ithaca, NY 14853, USA
gomes@cs.cornell.edu

**Resumen**

Muchos de los problemas que se plantean en Inteligencia Artificial pueden formalizarse como un problema de satisfacción de restricciones (CSP) y, luego, resolverse utilizando las técnicas de resolución que se han desarrollado para CSPs. En este artículo, empezamos definiendo el concepto de CSP y presentando algunos ejemplos de modelización de problemas combinatorios como CSPs. A continuación, describimos en detalle las técnicas de resolución de CSPs más utilizadas: algoritmos de propagación de restricciones (nodo consistencia, arco consistencia y *k*-consistencia), algoritmos de búsqueda (*generate and test*, *backtracking*, *backjumping* y *conflict-directed backjumping*) y algoritmos híbridos (*forward checking* y *maintaining arc consistency*).

## 1. Introducción

Muchos de los problemas que se plantean en Inteligencia Artificial pueden formalizarse como un problema de satisfacción de restricciones (CSP) y, luego, resolverse utilizando las técnicas de resolución que se han desarrollado para CSPs. Este método genérico de resolución de problemas ha demostrado ser altamente competitivo en áreas tan diversas como visión artificial, planificación, scheduling, razonamiento temporal y verificación de circuitos. Además, tiene la ventaja de que la modelización de los problemas es muy natural y declarativa, de manera que se formaliza lo que se ha de satisfacer sin necesidad de indicar cómo se tiene que satisfacer.

En este artículo, empezamos definiendo el concepto de CSP y presentando algunos ejemplos de modelización de problemas combinatorios como CSPs. A continuación, introducimos las técnicas de resolución de CSPs más utilizadas. En primer lugar, describimos algoritmos de propagación de restricciones; presentamos algoritmos para conseguir nodo consistencia, arco consistencia y *k*-consistencia. En segundo lugar, presentamos cuatro algoritmos de búsqueda: GT o algoritmo *generate and test*, BT o algoritmo *backtracking*, BJ o algoritmo *backjumping*, y CBJ o algoritmo *conflict-directed backjumping*. Finalmente, presentamos dos algoritmos híbridos: FC o algoritmo *forward checking*, y MAC o algoritmo *maintaining arc consistency*. Estos algoritmos introducen propagación de restricciones en los algoritmos de búsqueda y, en la práctica, son los más utilizados en la resolución de problemas computacionalmente difíciles.

A los lectores y a las lectoras que estén interesados en temas más avanzados, les remitimos a alguno de los pocos libros existentes sobre CSPs [8, 30], o bien a alguno de los surveys que se han publicado en los últimos años [2, 11, 19, 22, 24, 25]. Para conocer los últimos avances, remitimos a las actas del congreso anual sobre CSPs (*International Conference on Principles and Practice of Con-*

[*]Este trabajo ha sido financiado por el Ministerio de Ciencia y Tecnología, proyecto TIC2001-1577-C03-03.

*straint Programming*), que se publican en la serie *Lecture Notes in Computer Science* de Springer, y a la revista *Constraints*, publicada por Kluwer. En las actas de congresos y revistas generalistas de Inteligencia Artificial también se pueden encontrar trabajos sobre CSPs.

## 2. Problemas de satisfacción de restricciones

Un *problema de satisfacción de restricciones (CSP)* [26][1] se define por una terna $(X,D,R)$, donde

- $X = \{X_1, X_2, \ldots, X_n\}$ es un conjunto finito de *variables*,
- $D = \{D_1, D_2, \ldots, D_n\}$ es un conjunto finito de *dominios*, y
- $R = \{R_1, R_2, \ldots, R_r\}$ es un conjunto finito de *restricciones*.

Cada variable $X_i$ toma valores en su correspondiente dominio $D_i$. Una restricción entre un subconjunto de variables $\{X_{i_1}, \ldots, X_{i_k}\}$ de $X$ es un subconjunto del producto cartesiano $D_{i_1} \times \cdots \times D_{i_k}$ que está formado por las tuplas de valores que satisfacen la restricción; dicho de otra forma, contiene las combinaciones de asignaciones de valores a variables que no violan la restricción.

Una restricción en la que interviene una única variable es una restricción *unaria*, y una restricción en la que intervienen dos variables es una restricción *binaria*. Una restricción unaria en la que interviene la variable $X_i$ la representaremos por $R_i$, y una restricción binaria en la que intervienen las variables $X_i$ e $X_j$ la representamos por $R_{ij}$. Un *CSP binario* es un CSP que sólo tiene restricciones unarias o binarias.

En este artículo siempre haremos referencia a CSPs binarios y, además, supondremos que todos los dominios son de cardinalidad finita. De hecho, limitarnos a CSPs binarios no supone ninguna restricción, puesto que cualquier CSP puede convertirse en un CSP binario en tiempo polinomial [8].

Una asignación de un valor del dominio a cada variable de un CSP es una *solución* del CSP si dicha asignación satisface todas las restricciones.

[1] CSP es el acrónimo de *Constraint Satisfaction Problem.*

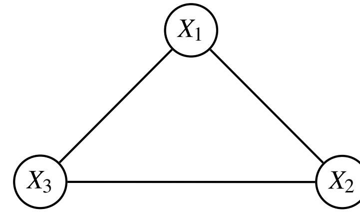



Figura 1: Ejemplo de grafo

Solucionar un CSP consiste en encontrar una solución, encontrar todas las soluciones, o encontrar una solución óptima con respecto a una función objetivo.

**Ejemplo 1** *El problema de la k-coloración de un grafo consiste en decidir si los nodos del grafo pueden colorearse empleando k colores de manera que dos nodos adyacentes no estén coloreados con el mismo color. Su representación como un CSP se puede definir de la siguiente forma:*

- *$X$ es el conjunto de nodos.*
- *$D$ asigna el dominio $\{1,2,\ldots,k\}$ a cada variable, donde cada elemento del dominio denota uno de los posibles colores.*
- *$R$ contiene la restricción $X_i \neq X_j$ para cada par $(X_i, X_j)$ de nodos adyacentes.*

*La formalización del problema de la 3-coloración para el grafo de la figura 1 como un CSP es la siguiente:*

- *Variables: $X = \{X_1, X_2, X_3\}$, una variable por nodo.*
- *Dominios: $D_i = \{azul, rojo, verde\}$, $1 \leq i \leq 3$, un valor por color.*
- *Restricciones: nodos adyacentes tienen colores diferentes.*

$$\begin{aligned}
R_{12} &= \{(azul, rojo), (azul, verde), \\
&\quad (rojo, verde)\} \\
R_{13} &= \{(azul, rojo), (azul, verde), \\
&\quad (rojo, verde)\} \\
R_{23} &= \{(azul, rojo), (azul, verde), \\
&\quad (rojo, verde)\}
\end{aligned}$$

**Ejemplo 2** *El problema de las n reinas consiste en colocar n reinas en un tablero de ajedrez de dimensiones $n \times n$ de forma que no se ataquen. Una*

*manera de representar este problema consiste en modelizar las filas del tablero como variables y las columnas como valores del dominio. En este caso, cuando decimos que hemos asignado el valor j ($1 \leq j \leq n$) a la variable $X_i$ ($1 \leq i \leq n$), estamos diciendo que hemos colocado una reina en la intersección de la fila i y la columna j. Para representar que dos reinas no pueden atacarse entre sí, necesitamos representar que no pueden colocarse ni en la misma columna ni en la misma diagonal. Para ello, añadimos la siguiente restricción por cada par de variables $X_i, X_j$:*

$$R_{ij} = \{(a,b) | a \neq b \wedge |i-j| \neq |a-b|\} \ (i > j)$$

*La formalización del problema de las 4 reinas como un CSP es la siguiente:*

- *Variables: $X = \{X_1, X_2, X_3, X_4\}$, una variable por fila.*
- *Dominios: $D_i = \{1,2,3,4\}$, $1 \leq i \leq 4$, un valor por columna.*
- *Restricciones: no colocar dos reinas ni en la misma columna ni en la misma diagonal.*

$$\begin{aligned}
R_{12} &= \{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)\} \\
R_{13} &= \{(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),\\
&\quad (4,1),(4,3)\} \\
R_{14} &= \{(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),\\
&\quad (3,2),(3,4),(4,2),(4,3)\} \\
R_{23} &= \{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)\} \\
R_{24} &= \{(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),\\
&\quad (4,1),(4,3)\} \\
R_{34} &= \{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)\}
\end{aligned}$$

*La figura 2 muestra un tablero que es solución al problema de las 4 reinas y un tablero que viola las restricciones $R_{14}$ y $R_{23}$.*

Un CSP binario se acostumbra a representar mediante un *grafo de restricciones*. Cada nodo del grafo corresponde a una variable del problema. Cada restricción se representa por una arista que tiene como etiqueta el nombre de la restricción. Si la restricción es unaria, la arista tiene como origen y final el mismo nodo. Si la restricción es binaria, conecta los nodos de las dos variables que intervienen en la restricción. Una arista que conecta dos nodos $X_i$ y $X_j$ representa dos arcos: el arco que va de $X_i$ a $X_j$ y el arco que va de $X_j$ a $X_i$.

| | | | | |
|---|---|---|---|---|
| $X_1$ | | $R$ | | |
| $X_2$ | | | | $R$ |
| $X_3$ | $R$ | | | |
| $X_4$ | | | $R$ | |

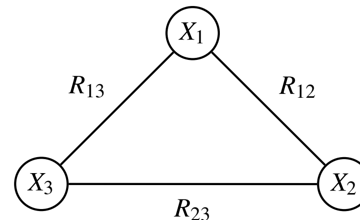
correcto

| | | | | |
|---|---|---|---|---|
| $X_1$ | $R$ | | | |
| $X_2$ | | | $R$ | |
| $X_3$ | | $R$ | | |
| $X_4$ | | | | $R$ |

incorrecto

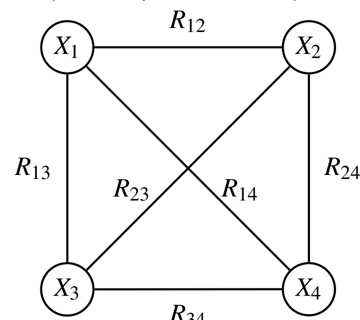
Figura 2: Problema de las 4-reinas

$D_1 = \{azul, rojo, verde\}$




$D_3 = \{azul, rojo, verde\}$ $D_2 = \{azul, rojo, verde\}$

$D_1 = \{1,2,3,4\}$ $D_2 = \{1,2,3,4\}$




$D_3 = \{1,2,3,4\}$ $D_4 = \{1,2,3,4\}$

Figura 3: Grafos de restricciones

**Ejemplo 3** *La figura 3 muestra los grafos de restricciones correspondientes al problema de la 3-coloración del grafo de la figura 1 y al problema de las 4 reinas. Junto a cada nodo, también mostramos el dominio de la variable que corresponde al nodo.*

# 3. Inferencia en CSPs

En esta sección describimos algoritmos de propagación de restricciones, que son aquellos que, a partir de un CSP $P$, crean otro CSP $P'$ *equivalente* (es decir, con el mismo conjunto de soluciones) que es más simple de resolver. El estudio de la inferencia tiene sus orígenes en los trabajos de Waltz sobre interpretación de escenas tridimensionales [31].

Los algoritmos que describimos —algoritmos de *nodo consistencia* y *arco consistencia*— obtienen $P'$ filtrando el dominio de las variables de $P$ mediante la eliminación de valores que no forman parte de ninguna solución. Puesto que el número de posibles asignaciones es la cardinalidad del producto cartesiano del dominio de todas las variables, cada vez que reducimos el dominio de una variable, estamos descartando asignaciones que no son solución y reducimos el número de posibles asignaciones a considerar.

Otra manera de simplificar —que no tratamos aquí y que se consigue con los llamados algoritmos de *camino consistencia* [20]— consiste en eliminar tuplas cuyas asignaciones no aparecen en ninguna solución y en derivar nuevas restricciones binarias que se siguen de las restricciones que tenemos. Por ejemplo, dado un CSP $P$ con las restricciones $X_1 < X_2$ y $X_2 < X_3$, podemos crear $P'$ añadiendo a $P$ la restricción $X_1 < X_3$.

Los algoritmos de propagación de restricciones que veremos son incompletos, por lo que suelen utilizarse en combinación con algoritmos de búsqueda, dando lugar a algoritmos híbridos. También pueden utilizarse como técnica de preprocesamiento antes de aplicar un algoritmo de búsqueda.

La técnica más simple de propagación de restricciones se denomina consistencia de nodo (NC). NC consiste en eliminar, en aquellas variables para las que tenemos definidas restricciones unarias, los valores del dominio que son inconsistentes con las restricciones unarias. En el resto del artículo, supondremos que todos nuestros CSPs son nodo consistentes.

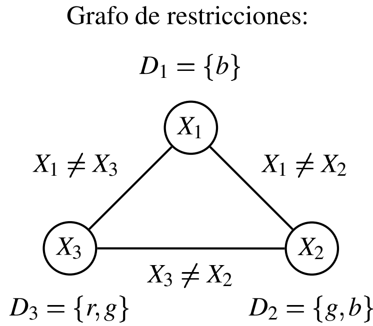
```
procedimiento AC-1
 entrada: un CSP P = (X, D, R)
 salida: un CSP equivalente y arco consistente
   Q ← {(i, j)|(X_i, X_j) es un arco del grafo de P, i ≠ j}
   repetir
     cambio ← FALSO
     para cada (i, j) ∈ Q hacer
       cambio ← revisar(i, j) o cambio
   hasta no cambio
fin procedimiento

funcion revisar(i, j)
   borrar ← FALSO
   para cada a ∈ D_i hacer
     si no existe b ∈ D_j tq. (a, b) ∈ R_ij
       D_i ← D_i − a
       borrar ← CIERTO
   retorna borrar
fin funcion
```

Figura 4: Algoritmo AC-1

La técnica de propagación de restricciones más popular, y que exige un mayor grado de consistencia que NC, es la conocida como consistencia de arcos (AC). AC se aplica a restricciones binarias. Dado un CSP binario y nodo consistente $P$, una restricción binaria $R_{ij}$ de $P$ es arco consistente direccional (de $i$ a $j$) si, y sólo si, para todo valor $a \in D_i$ existe un valor $b \in D_j$ tal que $(a,b) \in R_{ij}$. Una restricción $R_{ij}$ de $P$ es arco consistente si es arco consistente direccional en los dos sentidos. El CSP $P$ es arco consistente si, y sólo si, lo son todas sus restricciones.

La figura 4 muestra el pseudocódigo del algoritmo AC-1 [20]. Dado un CSP, AC-1 retorna un CSP equivalente y arco consistente. AC-1 emplea la función *revisar*, tantas veces como sea necesario, para conseguir que todos los arcos del grafo de restricciones sean arco consistentes. Para conseguir que un arco $(i, j)$ sea arco consistente, esta función elimina los valores del dominio de $i$ que no son compatibles con ningún valor del dominio de $j$. Los valores eliminados no pertenecen a ninguna solución y, por tanto, son redundantes. La función *revisar*$(i, j)$ retorna *cierto* si ha eliminado algún valor redundante; en caso contrario, el arco de entrada ya es arco consistente y retorna *falso*. Dado un CSP con $n$ variables, con dominios cuya máxima cardinalidad es $k$, y con $e$ restricciones, la complejidad de AC-1 es $O(enk^3)$. AC-1 es un algoritmo



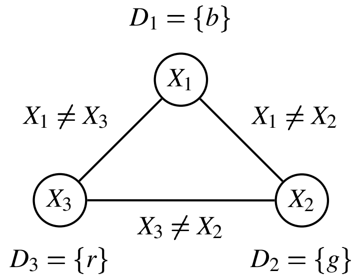



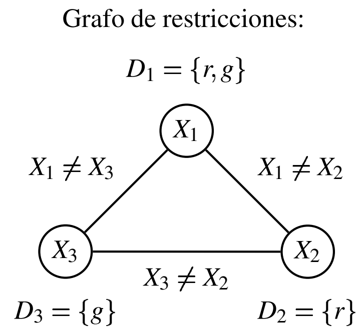

Figura 5: Obtención de una solución mediante consistencia de arcos

de fuerza bruta, pero hay algoritmos más eficientes, como AC-4 [21] y AC2001 [5], cuya complejidad —$O(ek^2)$— es óptima. Otro algoritmo muy utilizado es AC-3 [20].

En general, los algoritmos de consistencia de arco no permiten decidir si un CSP tiene solución, excepto en los siguientes casos: (i) si algún dominio queda vacío, entonces no hay ninguna solución; (ii) si todos los dominios contienen un único elemento, entonces tiene una solucion que es la que se obtiene al asignar a cada variable el valor de su dominio, o (iii) si una variable tiene un dominio con $m$ elementos y el resto de las variables tienen un dominio con un único elemento, entonces hay $m$ soluciones.

**Ejemplo 4** *La figura 5 muestra el grafo de restricciones de un CSP y el grafo que se obtiene al aplicarle el algoritmo AC-1. Puesto que todos los dominios tienen cardinalidad uno, hemos encontrado una solución.*

*La figura 6 muestra el grafo de restricciones de otro CSP y el grafo que se obtiene al aplicarle el algoritmo AC-1. Puesto que tenemos un dominio vacío, el problema no tiene solución.*



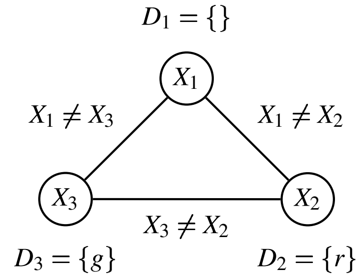
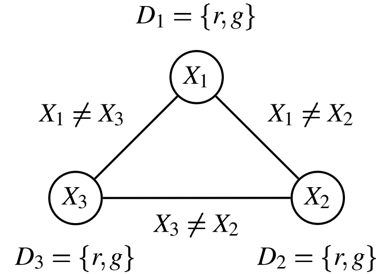




Figura 6: Detección de que no hay solución mediante consistencia de arcos



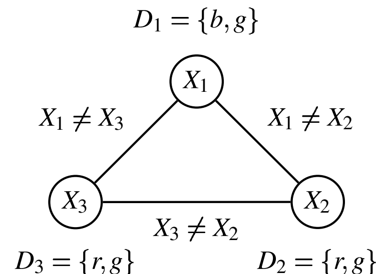




Figura 7: Dos CSPs para los que no podemos detectar nada mediante consistencia de arcos

*La figura 7 muestra el grafo de restricciones de dos CSPs; el primero no tiene solución y el segundo tiene dos soluciones. Sin embargo, no podemos concluir nada aplicando el algoritmo AC-1.*

Hay grados de consistencia más fuertes que los anteriores: $k$-consistencia [13] y $k$-consistencia fuerte [14]. Un grafo de restricciones es $k$-consistente si para cualquier subconjunto de $k-1$ variables que satisfacen todas las restricciones entre ellas, existe un valor $d \in D_k$ para cualquier otra variable $X_k$ de manera que se satisfacen todas las restricciones entre las $k$ variables. Un grafo es $k$-consistente fuerte si es $j$-consistente para todo $j \leq k$. NC es equivalente a 1-consistencia fuerte y AC a 2-consistencia fuerte.

Un CSP con $n$ variables se puede solucionar, sin necesidad de realizar búsqueda, consiguiendo $n$-consistencia. Sin embargo, esta técnica, que acostumbra a ser computacionalmente más costosa que otros métodos basados en búsqueda, no suele utilizarse. Normalmente, lo más útil es emplear consistencia de arco como técnica de propagación de restricciones para simplificar el problema. Como en la mayoría de los casos no es suficiente para obtener una solución, lo que se hace en la práctica es combinar propagación de restricciones con búsqueda.

Otras técnicas de inferencia que permiten solucionar CSPs sin búsqueda son la consistencia adaptativa [9], *tree clustering* [10], y *bucket elimination* [7]. En estas técnicas se realiza inferencia teniendo en cuenta un orden entre las variables. En general, resulta más eficiente solucionar un CSP con estas técnicas que solucionarlo con $n$-consistencia.

# 4. Búsqueda en CSPs

El conjunto formado por todas las posibles asignaciones de un CSP, también conocido como *espacio de estados*, se representa mediante un árbol de búsqueda. En cada nivel instanciamos una variable, y los sucesores de un nodo son todos los valores de la variable asociada a este nivel. Cada camino del nodo raíz a un nodo terminal, denominado *rama*, representa una *asignación completa*. La raíz, que corresponde al nivel 0, representa la asignación vacía, y cada camino del nodo raíz a un nodo no terminal representa una *asignación parcial*. El número total de ramas es la cardinalidad del producto cartesiano de los dominios de todas las variables.

A continuación presentamos cuatro algoritmos, que se diferencian en cómo recorren el árbol de búsqueda a la hora de encontrar soluciones: GT (*generate and test*), BT (*chronological backtracking*), BJ (*backjumping*) y CBJ (*conflict-directed backjumping*). Cuando creamos el árbol de búsqueda, denominamos *variable actual* a la que estamos asignando un valor, *variables pasadas* a las que ya tienen un valor asignado, y *variables futuras* a las que están sin asignar.

En la descripción de los algoritmos supondremos, para facilitar su comprensión, que el orden en el que se instancian las variables en cada rama es estático y sigue la numeración de los subíndices de las variables; es decir, instanciamos primero $x_1$, luego $x_2$, y así sucesivamente. A este orden se le denomina *orden lexicográfico*. No obstante, el rendimiento de los algoritmos puede mejorse mucho introduciendo alguna heurística de ordenación de variables dinámica.

Una de las heurísticas de selección de variable más exitosas consiste en instanciar primero aquellas variables que tienen menor dominio [6], rompiendo empates escogiendo la variable que aparece en más restricciones.

Una vez seleccionada la variable, debemos asignarle uno de los valores del dominio. El orden en que asignamos valores a variables también puede mejorar el rendimiento cuando nuestro objetivo es encontrar la primera solución. Una heurística popular de selección de valor consiste en asignar primero el valor que es consistente con mayor número de valores factibles del resto de las variables.

Los algoritmos de esta sección son completos, puesto que realizan un recorrido exhaustivo del espacio de estados. Cuando aplican alguna poda, lo hacen sobre regiones del espacio de estados que no contienen ninguna solución.

## 4.1. Algoritmo GT

La manera más sencilla, aunque poco eficiente, de encontrar todas las soluciones de un CSP es la que implementa el algoritmo GT (*generate-and-test*). GT genera, de forma sistemática, todas las posibles asignaciones completas. Cuando finaliza de generar una asignación completa, comprueba si esta asig-

```
procedimiento BT(i)
 entrada: un CSP P = (X,D,R) con n variables
 salida: las soluciones de P
  para cada a ∈ D_i hacer
    asignaciones[i] ← a
    consistente ← CIERTO
    para h ← 1 hasta i−1 mientras consistente
      consistente ← test(i,h)
    si consistente
      si i = n
        mostrar-solucion()
      sino
        BT(i+1)
  retorna FALSO
fin procedimiento
```

Figura 8: Algoritmo BT

nación es una solución (es decir, comprueba si satisface todas las restricciones). En terminología de árboles, GT es un algoritmo que recorre el árbol de búsqueda en profundidad prioritaria (recorrido primero en profundidad). La ineficiencia de GT se debe a que genera muchas asignaciones completas que violan la misma restricción.

## 4.2. Algoritmo BT

El algoritmo BT (*chronological backtracking*) [6] mejora el algoritmo GT de la siguiente forma: cada vez que se asigna un nuevo valor a la variable actual ($X_i$), se comprueba si es consistente con los valores que hemos asignado a las variables pasadas. Si no lo es, se abandona esta asignación parcial, y se asigna un nuevo valor a $X_i$. Si ya se han agotado todos los valores de $D_i$, BT retrocede para probar otro valor para la variable $X_{i-1}$. Si se ha agotado $D_{i-1}$, retrocede al nivel $i-2$, y así sucesivamente hasta encontrar una asignación de un valor a una variable que es consistente con las variables pasadas o hasta que se demuestra que no hay más soluciones. Es decir, BT recorre el árbol utilizando búsqueda primero en profundidad y, en cada nodo, comprueba si la variable actual es consistente con las variables pasadas. Si detecta inconsistencia, descarta la asignación parcial actual, puesto que no es parte de ninguna asignación completa que sea solución. De esta forma, se ahorra recorrer el subárbol que *cuelga* de esta asignación parcial.

La figura 8 muestra el pseudocódigo de BT [25], utilizando la siguiente notación:

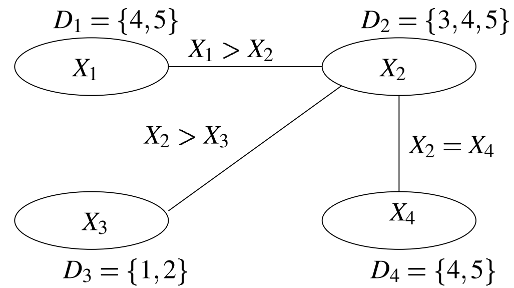



Figura 9: Grafo de restricciones para un CSP

- $i$: el parámetro de entrada $i$ es la posición de la variable actual en el orden estático en el que se instancian las variables.
- $n$: es el número de variables del problema.
- *asignaciones*[]: es un array de tamaño $n$ que guarda el valor asignado a cada variable.
- $test(i,j)$: devuelve *cierto* si la asignación actual no viola ninguna restricción entre las variables $X_i$ y $X_j$.

Por simplicidad, el pseudocódigo no distingue entre no encontrar solución y no encontras más soluciones; devuelve *falso* en ambos casos.

BT tiene el inconveniente de que detecta, en diferentes partes del espacio de búsqueda, inconsitencias que se producen por la misma razón. Este fenómeno se conoce como *trashing*. Por ejemplo, supongamos que tenemos una restricción entre la variable $X_g$ y la variable $X_i$, donde $i > g+1$, que establece que si asignamos a $X_g$ un determinado valor $a$, entonces ningún valor de $X_i$ es compatible con la asignación $X_g \leftarrow a$. Entonces, BT detecta, al nivel $i$, una inconsistencia en todas las asignaciones parciales que contienen la asignación $X_g \leftarrow a$. Cuando detecta una de estas inconsistencias, BT retrocede al nivel $i-1$, cuando en realidad podría retroceder al nivel $g$ y asignar un nuevo valor a $X_g$. Obsérvese que si BT retrocede al nivel $i-1$, volverá a detectar inconsistencia, para cualquier valor de $D_{i-1}$, cuando instancie la variable $X_i$, puesto que todas las asignaciones parciales que considera mantienen la asignación $X_g \leftarrow a$. Esto sucederá para todos los niveles intermedios entre $g$ e $i$. Por tanto, se podría podar el subárbol que cuelga de la asignación parcial que contiene $X_g \leftarrow a$.

Para solucionar este problema, se han diseñado algoritmos que aplican backtracking no cronológi-

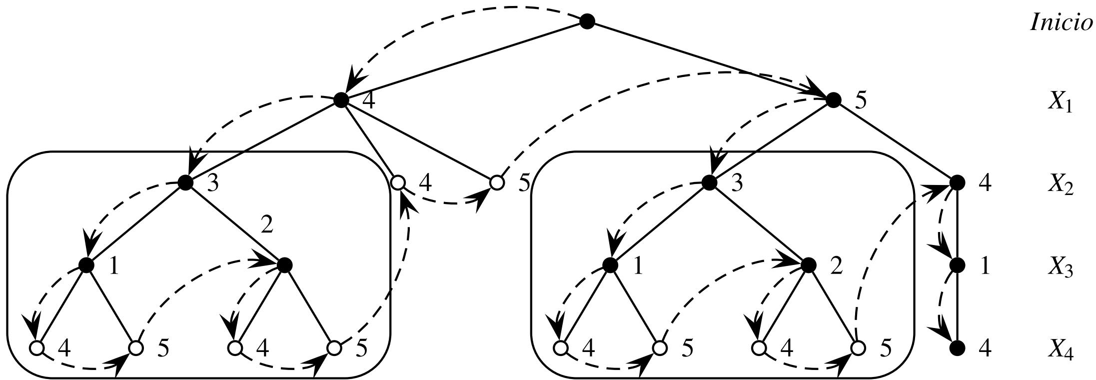



Figura 10: Árbol de búsqueda que explora BT para el CSP de la figura 9

co. Cuando se detecta una inconsistencia, en lugar de retroceder a la variable más recientemente instanciada que todavía tiene alternativas disponibles, se continúa la búsqueda en aquella variable que causó el fallo, evitando así que se repita la misma inconsistencia.

**Ejemplo 5** *La figura 10 muestra el árbol de búsqueda que explora BT para encontrar la primera solución del CSP de la figura 9. Cada rama representa una asignación parcial (que es completa si llegamos a instanciar la variable $X_4$). Cada nodo tiene como etiqueta el valor del dominio que asignamos a la variable; los nodos huecos representan asignaciones en las que la asignación de la variable actual es inconsistente con las asignaciones de las variables pasadas que aparecen en la misma rama. El comportamiento de* trashing *corresponde a las regiones que hemos insertado en un recuadro.*

## 4.3. Algoritmo BJ

El algoritmo BJ (*backjumping*) [16] es un algoritmo de backtracking no cronológico que mitiga el *trashing*. Su pseudocódigo [25] se muestra en la figura 11. BJ recorre el árbol como BT, pero cuando detecta que la asignación que hemos hecho a la variable actual $X_i$ es inconsistente con la asignación que tenemos, en la misma rama, para alguna variable pasada $X_h$, guarda el nivel de esta variable (en el pseudocódigo, asigna $h$ a la variable *max_nivel_retroceso*); es decir, el algoritmo recuerda en qué nivel se ha detectado una inconsistencia.

**procedimiento** BJ($i$)
 **entrada:** un CSP $P = (X, D, R)$ con $n$ variables
 **salida:** las soluciones de P
  max_nivel_retroceso $\leftarrow$ 0
  profundidad_retroceso $\leftarrow$ 0
  **para cada** $a \in D_i$ **hacer**
   asignaciones[$i$] $\leftarrow$ a
   consistente $\leftarrow$ CIERTO
   **para** $h \leftarrow 1$ **hasta** $i-1$ **mientras** consistente
    consistente $\leftarrow$ test$(i,h)$
   max_nivel_retroceso $\leftarrow h-1$
   **si** consistente
    **si** $i = n$
     mostrar-solucion()
    **sino**
     max_nivel_retroceso $\leftarrow$ BJ$(i+1)$
     **si** max_nivel_retroceso $< i$
      **retorna** max_nivel_retroceso
   profundidad_retroceso $\leftarrow$
    máx(profundidad_retroceso, max_nivel_retroceso)
  **retorna** profundidad_retroceso
**fin procedimiento**

Figura 11: Algoritmo BJ

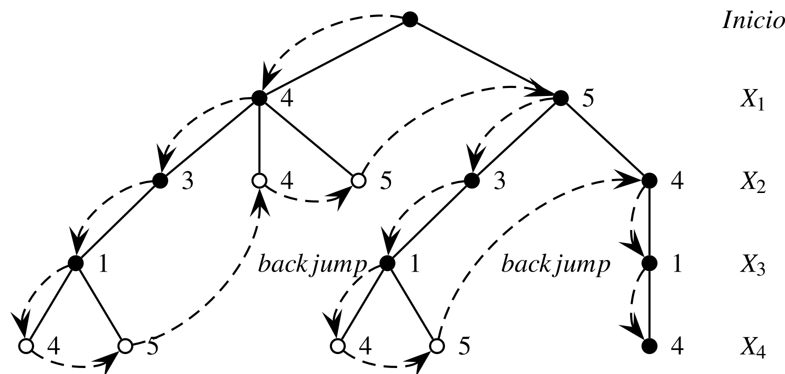



Figura 12: Árbol de búsqueda que explora BJ para el CSP de la figura 9

En caso de que no se haya detectado ninguna inconsistencia, se guarda el valor $i-1$. A continuación, BJ asigna a $X_i$ un nuevo valor de $D_i$, y también guarda el nivel en el que ha detectado una inconsistencia. Cuando se agota $D_i$, BJ retrocede al máximo nivel de retroceso que hemos detectado para la variable $X_i$ (en el pseudocódigo, este valor se guarda en la variable *profundidad_retroceso*), en lugar de retroceder a la variable más recientemente instanciada como en BT. El hecho de cambiar la instanciación de $X_h$ nos puede permitir encontrar una instanciación que sea consistente con $X_i$. Sin embargo, cambiar la instanciación de cualquiera de las variables que hay entre $X_i$ y $X_h$ és inútil, puesto que no ha cambiado la razón por la que hemos detectado un bloqueo para $X_i$.

**Ejemplo 6** *La figura 12 muestra el árbol de búsqueda que explora BJ para el CSP de la figura 9. El comportamiento de* trashing *que se observa en la figura 10 se evita con el algoritmo BJ.*

## 4.4. Algoritmo CBJ

El algoritmo CBJ (*conflict-directed backjumping*) [28] tiene un mecanismo de backtracking no cronológico más sofisticado que BJ. Si BJ retrocede a un nivel ($l$) y resulta que ya hemos probado todos los valores del dominio de la variable $X_l$, BJ retrocede al nivel $l-1$, que es la profundidad de retroceso asociada a $l$. Sin embargo, en algunos casos, es posible retroceder más atrás. Esto se evita con CBJ, cuyo pseudocódigo [25] se muestra en la figura 13.

```
procedimiento CBJ(i)
 entrada: un CSP P = (X,D,R) con n variables
 salida: las soluciones de P
   cjto_conflicto[i] ← 0
   para cada a ∈ D_i hacer
     asignaciones[i] ← a
     consistente ← CIERTO
     para h ← 1 hasta i − 1 mientras consistente
       consistente ← test(i,h)
     si no consistente
       cjto_conflicto[i] ← cjto_conflicto[i] ∪ {h − 1}
     si consistente
       si i = n
         mostrar-solucion()
         cjto_conflicto[i] ← cjto_conflicto[i] ∪ {n − 1}
       sino
         profundidad_retroceso ← CBJ(i + 1)
         si profundidad_retroceso < i
           retorna profundidad_retroceso
   profundidad_retroceso ← máx(cjto_conflicto[i])
   cjto_conflicto[profundidad_retroceso] ←
     cjto_conflicto[profundidad_retroceso] ∪
     cjto_conflicto[i] \ {profundidad_retroceso}
   retorna profundidad_retroceso
fin procedimiento
```

Figura 13: Algoritmo CBJ

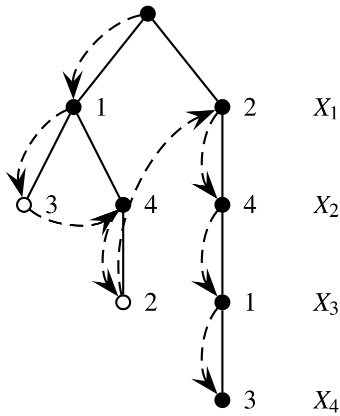



Figura 14: Árbol generado por FC para el problema de las 4 reinas

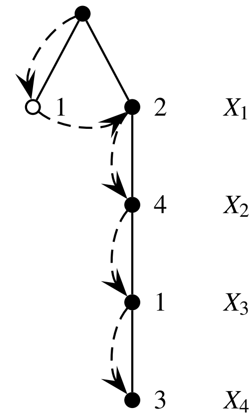



Figura 15: Árbol generado por MAC para el problema de las 4 reinas

En CBJ, asociamos a cada variable $X_i$ su *conjunto conflicto*, que contiene las variables pasadas para las cuales se ha detectado alguna inconsistencia con $X_i$. Cada vez que se detecta una inconsistencia entre una instanciación $a_i$ de la variable actual $X_i$ y una instanciación $a_h$ de alguna variable pasada $X_h$, se añade la variable $X_h$ al conjunto conflicto de $X_i$. Cuando se ha agotado el dominio de $X_i$, CBJ retrocede a la variable más profunda ($X_g$) del conjunto conflicto de $X_i$. Al mismo tiempo, las variables del conjunto conflicto de $X_i$, con la excepción de $X_g$, se añaden al conjunto conflicto de $X_g$, de manera que no se pierde información sobre conflictos.

En cada invocación de CBJ($i$), el conjunto conflicto de $X_i$ se vacía de cualquier información que contenga de invocaciones previas. Si CBJ encuentra una solución, se añade al conjunto conflicto de $X_n$ la variable $X_{n-1}$ para garantizar que CBJ retrocederá al nivel $n-1$ cuando se hayan explorado todos los valores de $D_n$.

## 5. Algoritmos Híbridos

Finalmente, presentamos dos algoritmos híbridos: FC o algoritmo *forward checking*, y MAC o algoritmo *maintaining arc consistency*. Estos algoritmos introducen propagación de restricciones en el algoritmo BT y, en la práctica, son los más utilizados en la resolución de problemas computacionalmente difíciles.

FC [17] es un algoritmo que fuerza a que, en cada nodo, no haya ninguna variable futura que tenga algún valor de su dominio que sea inconsistente con el valor que alguna variable pasada tiene asignado en la rama que estamos considerando. Para conseguirlo, FC elimina los valores de los dominios de variables futuras que no cumplen esta condición; si alguno de estos dominios queda vacío, FC hace backtracking. Esta condición es equivalente a exigir que sea arco consistente el conjunto de restricciones en las que interviene la variable actual y una variable futura. Es decir, FC funciona como BT, pero realiza arco consistencia entre restricciones que conectan la variable actual y una variable futura cada vez que se asigna un valor a la variable actual.

MAC [29] funciona como FC, pero en cada nodo del árbol de búsqueda se aplica el algoritmo AC para alcanzar arco consistencia entre *todas* las restricciones del problema. Es decir, en cada nodo, se simplifica mediante arco consistencia el CSP que estamos considerando. Mientras que en FC exigimos arco consistencia parcial, en MAC exigimos arco consistencia total en cada nodo. Existen otros algoritmos (por ejemplo, *partial lookahead* y *full lookahead*) que, en cada nodo, exigen un grado de consistencia local intermedio entre FC y MAC [17].

**Ejemplo 7** *La figura 14 muestra el árbol generado por FC para encontrar la primera solución del problema de las 4 reinas (suponiendo que las variables se instancian en orden lexicográfico), y la figura 15 muestra el árbol generado por MAC. MAC genera 3 nodos menos que FC. Para conseguirlo, realiza mayor procesamiento por nodo. La solución generada es la que se muestra en la figura 2.*

Si aplicamos en cada nodo del árbol de búsqueda generado por CBJ la misma propagación de restricciones que FC y MAC aplican a cada nodo del árbol de búsqueda generado por BT, obtenemos dos nuevos algoritmos híbridos, que se conocen como CBJ-FC y CBJ-MAC.

El nivel de consistencia óptimo que hay que exigir, en cada nodo del árbol de búsqueda generado por un algoritmo híbrido, depende de la estructura del CSP que queremos resolver. Para cada CSP hay que encontrar el compromiso entre lo que ganamos al podar subárboles del espacio de búsqueda y el coste computacional que supone exigir un cierto grado de consistencia en cada nodo del árbol de búsqueda. Una comparación teórica de diferentes algoritmos híbridos puede encontrarse en [18]. Una comparación experimental exhaustiva de algoritmos que resuelven CSPs no se ha realizado; algunos trabajos que contienen comparaciones experimentales son [4, 15, 17, 27, 29].

# 6. Conclusiones

En este artículo hemos presentado las principales técnicas de resolución de CSPs basadas tanto en inferencia como en búsqueda, así como algoritmos híbridos que combinan inferencia y búsqueda. Con ello, esperamos haber proporcionado una primera aproximación al tema de CSPs y haber sentado las bases para abordar temas más avanzados tales como *soft constraints* [23], resolución de CSPs no binarios [1, 3] y CSPs distribuidos [12, 32].

# Referencias


[1] F. Bacchus, X. Chen, P. van Beek, y T. Walsh. Binary vs non-binary constraints. *Artificial Intelligence*, 140:1–37, 2002.

[2] R. Barták. Theory and practice of constraint propagation. En *Proceedings of 3rd Workshop on Constraint Programming for Decision and Control*, pages 7–14, 2001.

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

[4] C. Bessière y J. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. En *2nd International Conference on Principles and Practice of Constraint Programming, CP-96*, pages 61–75. Springer LNCS 1118, 1996.

[5] C. Bessière y J. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, 2001.

[6] J. Bitner y E. Reingold. Backtracking programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[7] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.

[8] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

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

[10] R. Dechter y J. Pearl. Tree-clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[11] R. Dechter y F. Rossi. Constraint satisfaction. En L. Nadel, editor, *Encyclopedia of Cognitive Science*. Nature/Scientific American Publishing Group, 2002.

[12] C. Fernàndez, R. Béjar, B. Krishnamachari, y C. Gomes. Communication and computation in distributed CSP algorithms. En *8th International Conference on Principles and Practice of Constraint Programming, CP-2002*, pages 664–679. Springer LNCS 2470, 2002.

[13] E. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[14] E. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[15] D. Frost y R. Dechter. Looking at full looking ahead. In *2nd International Conference on Principles and Practice of Constraint Programming, CP-96*, pages 539–540. Springer LNCS 1118, 1996.

[16] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical report, CMU-CS-79-124, Carnegie Mellon University, 1979.

[17] R. Haralick y G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[18] G. Kondrak y P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[19] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[20] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[21] A. Mackworth. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[22] P. Meseguer. Constraint satisfaction problems: An overview. *AI Communications*, 2(1):3–17, 1989.

[23] P. Meseguer, N. Bouhmala, T. Bouzoubaa, M. Irgens, y M. Sánchez. Current approaches for solving over-constrained problems. *Constraints*, 8(1):9–39, 2003.

[24] I. Miguel y Q. Shen. Solution techniques for constraint satisfaction problems: Advanced approaches. *Artifical Intelligence Review*, 15:269–293, 2001.

[25] I. Miguel y Q. Shen. Solution techniques for constraint satisfaction problems: Foundations. *Artifical Intelligence Review*, 15:243–267, 2001.

[26] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7(66):95–132, 1974.

[27] B. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. En L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.

[28] P. Prosser. Hybrid algorithms for constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, 1993.

[29] D. Sabin y E. Freuder. Contradicting conventional wisdom in constraint satisfaction. En *Proceedings of ECAI'94*, pages 125–129, 1994.

[30] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[31] D. Waltz. Understanding line drawings of scenes with shadows. En P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[32] M. Yokoo y K. Hiramaya. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):198–212, 2000.