Capítulo 10

Problemas de satisfacción de restricciones (CSP)

10.1 Introducción

La programación por restricciones es una metodología software utilizada para la descripción y posterior resolución efectiva de cierto tipo de problemas, típicamente combinatorios y de optimización. Estos problemas aparecen en muy diversas áreas, incluyendo inteligencia artificial, investigación operativa, bases de datos y sistemas de recuperación de la información, etc., con aplicaciones en scheduling, planificación, razonamiento temporal, diseño en la ingeniería, problemas de empaquetamiento, criptografía, diagnóstico, toma de decisiones, etc. Estos problemas pueden modelarse como problemas de satisfacción de restricciones (*Constraint Satisfaction Problems* - CSP) y resolverse usando técnicas de satisfacción de restricciones. En general, se trata de grandes y complejos problemas, típicamente de complejidad NP. Las etapas básicas para la resolución de un problema CSP son su modelización y su posterior resolución mediante la aplicación de técnicas CSP específicas, que incluyen procesos de búsqueda apoyados con métodos heurísticos y procesos inferenciales. En este capítulo se presentan los conceptos, algoritmos y técnicas más relevantes en el área de los CSP, junto con diversos ejemplos y ejercicios.

Muchas decisiones que tomamos a la hora de resolver diversos problemas cotidianos están sujetas a restricciones. Decisiones tan cotidianas como fijar una cita, planificar un viaje, comprar un coche o preparar un plato de cocina puede depender de muchos aspectos interdependientes e incluso conflictivos, cada uno de los cuales está sujeto a un conjunto de restricciones que se deben satisfacer para que la decisión sea válida. Además, cuando se encuentra una solución que satisface plenamente a unos criterios, puede que no sea tan apropiada para otros, por lo que, para obtener una solución optimizada, no suele ser suficiente con obtener una única solución. Los primeros trabajos relacionados con la programación de restricciones datan de los años 60 y 70 en el campo de la Inteligencia Artificial. Durante los últimos años, la programación de

restricciones ha generado una creciente expectación entre expertos de muchas áreas debido a su potencial para la resolución de grandes y complejos problemas reales. Sin embargo, al mismo tiempo, se considera como una de las tecnologías menos conocida y comprendida. La programación de restricciones se define como el estudio de sistemas computacionales basados en restricciones. La idea de la programación de restricciones es resolver problemas mediante la declaración de restricciones sobre el dominio del problema y consecuentemente encontrar soluciones a instancias de los problemas de dicho dominio que satisfagan todas las restricciones y, en su caso, optimicen unos criterios determinados.

"Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it" (E. Freuder)

"Constraint Programming: A simple but powerful idea" (R. Dechter)

10.2 Definiciones y conceptos básicos

La programación de restricciones puede dividirse en dos ramas claramente diferenciadas: la "satisfacción de restricciones" y la "resolución de restricciones". Ambas comparten la misma terminología, pero sus orígenes y técnicas de resolución son diferentes. La satisfacción de restricciones trata con problemas que tienen dominios finitos, mientras que la resolución de restricciones está orientada principalmente a problemas sobre dominios infinitos o dominios más complejos. En este capítulo, se tratarán principalmente los problemas de satisfacción de restricciones (CSP). Los conceptos clave en esta metodología corresponden a los aspectos de:

- La modelización del problema, que permite representar un problema mediante un conjunto finito de variables, un dominio de valores finito para cada variable y un conjunto de restricciones que acotan las combinaciones válidas de valores que las variables pueden tomar. En la modelización CSP, es fundamental la capacidad expresiva, a fin de poder captar todos los aspectos significativos del problema a modelar.

- Técnicas inferenciales que permiten deducir nueva información sobre el problema a partir de la explícitamente representada. Estas técnicas también permiten acotar y hacer más eficiente el proceso de búsqueda de soluciones.

- Técnicas de búsqueda de la solución, apoyadas generalmente por criterios heurísticos, bien dependientes o independientes del dominio. El objetivo es encontrar un valor para cada variable del problema de manera que se satisfagan todas las restricciones del problema. En general, la obtención de soluciones en un CSP es NP-completo, mientras que la obtención de soluciones optimizadas es NP-duro, no existiendo forma de verificar la optimalidad de la solución en tiempo polinomial. Por ello, se requiere una gran eficiencia en los procesos de búsqueda.

10.2.1 Definición de un Problema de Satisfacción de Restricciones

Un problema de satisfacción de restricciones puede ser representado mediante una terna (X, D, C) donde:

- X es un conjunto de n variables $\{x_1, \ldots, x_n\}$.
- $D =< D_1, ..., D_n >$ es un tupla de dominios finitos donde se interpretan las variables X, tal que la i-ésima componente D_i es el dominio que contiene los posibles valores que pueden asignarse a la variable x_i. La cardinalidad de cada dominio es $d_i = |D_i|$.
- $C = \{c_1, c_2, \ldots, c_p\}$ es un conjunto finito de restricciones. Cada restricción k-aria c_i está definida sobre un conjunto de k variables $var(c_i) \subseteq X$, denominado su ámbito, y restringe los valores que dichas variables pueden simultáneamente tomar. Particularmente, una restricción es binaria cuando relaciona únicamente a dos variables $x_i y x_j$, y se suele denotar como c_{ij}. Todas las restricciones definidas en un CSP son conjuntivas, de manera que una solución debe de satisfacer a todas ellas.

La *instanciación* de una variable es un par variable-valor (x, a) que representa la asignación del valor a a la variable x $(x = a)$. La instanciación de un conjunto de variables es una tupla de pares ordenados, donde cada par ordenado (x_i, a_i) asigna el valor $\{a_i \in D_i\}$ a la variable x_i. Una tupla $((x_1, a_1), \ldots, (x_i, a_i))$ es localmente consistente si satisface todas las restricciones formadas por variables $\{x_1, .., x_i\}$ de la tupla. Para simplificar la notación, sustituiremos la tupla $((x_1, a_1), \ldots, (x_i, a_i))$ por $(a_1, \ldots, a_i)$.

Un valor $a_i \in D_i$ es un *valor consistente* para x_i si existe al menos una solución del CSP en la cual $x_i = a_i$. El dominio mínimo de una variable x_i es el conjunto de todos los valores consistentes para la variable, es decir, quedan excluidos aquellos valores que no forman parte de ninguna solución ($\forall x_i \in X, \forall a \in D_i, x_i = a$ forma parte de una solución del CSP).

Una *solución* a un CSP es una asignación $(a_1, a_2, \ldots, a_n)$ de valores a todas sus variables, de tal manera que se satisfagan todas las restricciones del CSP. Es decir, una solución es una tupla consistente que contiene todas las variables del problema. Una solución parcial es una tupla consistente que contiene algunas de las variables del problema. Un CSP es *consistente*, si tiene al menos una solución, es decir una tupla consistente. Dos CSPs son *equivalentes* si ambos representan el mismo conjunto de soluciones.

Ejemplo 10.1. *Para el problema de las 4-Reinas, existen solamente dos soluciones, que se detallan en la Figura 10.1. Asumiendo que las variables* $\{x_1, x_2, x_3, x_4\}$ *representan las columnas, y sus dominios son las posibles filas* $\{1, 4\}$ *donde colocar las reinas, tenemos que:* $(x_1, 3)$ *es una instanciación de la variable* x_1*, y* $((x_1, 2), (x_2, 4), (x_3, 1), (x_4, 3))$ *es una solución del CSP, por lo que el CSP es consistente. El valor 3 es un valor consistente para* x_1*, pero el valor 1 no lo es. El dominio mínimo de* x_2 *es* $\{1, 4\}$.

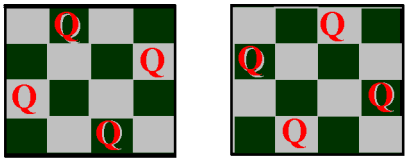


Figura 10.1: Dos soluciones al problema de las cuatro reinas.

Una vez modelado el problema como un CSP, los objetivos del CSP consisten en la *satisfabilidad* del mismo, es decir, obtener una o varias soluciones, sin preferencia alguna, o bien obtener una solución óptima, o al menos una buena solución, en base a una función objetivo previamente definida en términos de algunas o todas las variables. En este caso particular, se trata de un Problema de Satisfacción y Optimización de Restricciones, o por su acrónimo en inglés, CSOP (*Constraint Satisfaction and Optimization Problem*).

10.2.2 Definición y Tipología de las Restricciones.

Las restricciones se caracterizan fundamentalmente por su *aridad*, que es el número de variables involucradas en dicha restricción. Una restricción unaria es una restricción sobre una sola variable. Una restricción binaria es una restricción que consta de dos variables. Una restricción no binaria (o n-aria) es una restricción que involucra a un número arbitrario de 3 o más variables.

Ejemplo 10.2. *La restricción* $x \leq 5$ *es una restricción unaria sobre la variable* x*. La restricción* $x_4 - x_3 \neq 3$ *es una restricción binaria. La restricción* $2x_1 - x_2 + 4x_3 \geq 4$ *es una restricción ternaria. Por último, un ejemplo de restricción* n*-aria sería* $x_1 + 2x_2 - x_3 + 5x_4 \leq 9$*.*

Una restricción sobre un conjunto de variables puede definirse *extensionalmente* mediante un conjunto de tuplas válidas o no válidas y también *intensionalmente* mediante una función aritmética. La representación extensional de una restricción k-aria está formada por un conjunto de tuplas, cada una con k elementos, y expresa el conjunto de valores que las k variables pueden tomar simultáneamente. Claramente, en el caso de CSP continuos es imposible representar las restricciones extensionalmente ya que generalmente hay un número infinito de tuplas válidas.

Ejemplo 10.3. *Consideremos una restricción que involucra a las variables* x_1, x_2, x_3, x_4*, con dominios* $\{1, 2\}$*, donde la suma entre las variables* x_1 *y* x_2 *es menor o igual que la suma entre* x_3 *y* x_4*. Esta restricción puede representarse intensionalmente mediante la expresión* $x_1 + x_2 \leq x_3 + x_4$*. También, podría representarse extensionalmente mediante el conjunto de tuplas permitidas* $\{(1,1,1,1), (1,1,1,2), (1,1,2,1), (1,1,2,2), (2,1,2,2), (1,2,2,2), (1,2,1,2), (1,2,2,1), (2,1,1,2), (2,1,2,1), (2,2,2,2)\}$*.*

En el caso de una definición intensional, podemos tener restricciones *no-disyuntivas* o *disyuntivas*, según expresen una única relación o más de una relación disyuntiva

entre las variables. Por ejemplo, asumiendo un modelo con las relaciones elementales $\{<, =, >\}$, $(x_1 < x_2)$ es una restricción no-disyuntiva, mientras que $(x_1 < x_2 \vee x_1 > x_2)$, es decir, $x_1 \neq x_2$, sería una restricción disyuntiva. Por otra parte, las restricciones también pueden ser *cualitativas*, cuando expresan una relación de orden entre las variables (por ejemplo, $x_1 < x_2$), o *métricas* cuando expresan una distancia métrica entre las mismas (por ejemplo, $x_1 < x_2 + 7$). Las restricciones métricas requieren de una métrica en el dominio de interpretación de las variables.

Un tipo especial de restricciones son las restricciones lineales. Una *relación lineal* sobre $X = \{x_1, \ldots, x_k\}$ es una expresión de la forma:

$$\sum_{i=1}^{k} p_i x_i \, \{<, \leqslant, \neq, \geqslant, >\} \, b$$

Donde p_i son los coeficientes y $b \in \mathbb{R}$. En base a combinaciones lógicas de desigualdad ($\neq$) e inigualdad ($\leq$), se pueden expresar todas las relaciones en $\{<, \leqslant, \neq, \geqslant, >\}$. Las *Restricciones Lineales Disyuntivas* (*DLR*) son disyunciones de restricciones lineales.

10.3 Ejemplos de CSP y su modelización

El primer paso en la resolución de un CSP es su modelización, es decir, su representación en términos de variables, dominios y restricciones. Al igual que ocurre con el lenguaje natural, la modelización de un problema se puede realizar de muchas maneras diferentes. Respecto a la modelización de un problema CSP hay dos aspectos básicos:

- La potencia expresiva de las restricciones, es decir, la capacidad de modelar las restricciones realmente existentes en el problema real.
- La eficiencia de la representación, ya que dependiendo de la modelización CSP, el problema se resolverá con más o menos eficiencia.

10.3.1 Coloración del mapa.

Este problema parte de un conjunto de colores posibles para colorear cada región del mapa, de manera que regiones adyacentes tengan distintos colores. En la formulación del CSP, definimos una variable por cada región del mapa, y el dominio de cada variable es el conjunto de colores disponible. Para cada par de regiones contiguas existe una restricción sobre las variables correspondientes que no permite la asignación de idénticos valores a las variables. En el caso de la Figura 10.2, tenemos un mapa con cuatro regiones *x,y,z,w* para ser coloreadas con los posibles colores Rojo, Verde, Azul. La formulación CSP sería:

- Variables: $\{x, y, z, w\}$
- Dominio: $\{Rojo, Verde, Azul\}$, único para las tres variables.

- Restricciones (definición intensional): $\{x \neq y, x \neq w, x \neq z, y \neq w, w \neq z\}$

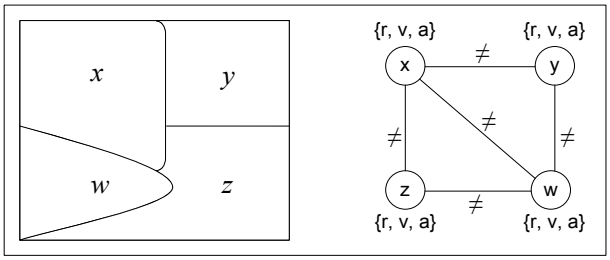


Figura 10.2: El problema de coloración del mapa.

Un CSP binario, puede ser representado mediante una red de restricciones, donde los nodos representan las variables y los arcos representan las restricciones entre las mismas. En la Figura 10.2-derecha, se representa la red correspondiente al problema del ejemplo, donde las variables correspondientes a regiones adyacentes están conectadas por una arista. Hay cinco restricciones en el problema, es decir, cinco aristas en la red. Una solución para el problema es la asignación $(x, r), (y, v), (z, v), (w, a)$. En esta asignación, todas las variables adyacentes tienen valores diferentes.

10.3.2 Criptografía

El típico problema criptográfico "*send+more=money*" consiste en asignar a cada letra $\{s, e, n, d, m, o, r, y\}$ un dígito diferente del conjunto $\{0, \ldots, 9\}$ de forma que se satisfaga: *send + more = money.*

	s	e	n	d
+	m	o	r	e
m	o	n	e	y

La manera más fácil de modelar este problema es asignando una variable a cada una de las letras, todas ellas con un dominio $\{0, \ldots, 9\}$ y con las restricciones que obliguen a que todas las variables tomen valores distintos y con la correspondiente restricción para que se satisfaga "*send + more = money*". De esta forma las restricciones son:

- $10^3(s + m) + 10^2(e + o) + 10(n + r) + d + e = 10^4m + 10^3o + 10^2n + 10e + y$;
- restricción de todas diferentes: $\neq (s, e, n, d, m, o, r, y)$;

La restricción de "todas diferentes" puede reemplazarse por un conjunto de restricciones binarias, $\{s \neq e, s \neq n, \ldots, r \neq y\}$, donde $(x_i \neq x_j)$ es la relación disyuntiva $x_i \{<, >\} x_j$.

Sin embargo, para el algoritmo de resolución más general como es Backtracking (que veremos en las siguientes secciones), este modelo no es muy eficiente porque todas

las variables necesitan ser instanciadas antes de comprobar estas dos restricciones. De esta manera no se podría podar el espacio de búsqueda durante el propio proceso a fin de agilizar la búsqueda. Además, la primera restricción es una igualdad en la que forman parte todas las variables del problema (restricción global) por lo que dificulta el proceso de consistencia. Un modelo más eficiente podría utilizar los bits de acarreo para descomponer la ecuación anterior en un conjunto de pequeñas restricciones. Esto requeriría incluir tres variables *portadoras* adicionales, c_1, c_2, c_3, cuyo dominio es $\{0, 1\}$:

- $e + d = y + 10c_1$;
- $c_1 + n + r = e + 10c_2$;
- $c_2 + e + o = n + 10c_3$;
- $c_3 + s + m = 10m + o$
- restricción de todas diferentes: $\neq (s, e, n, d, m, o, r, y)$

La ventaja de este modelo es que estas restricciones con menor aridad, puedan comprobarse antes en la búsqueda de backtracking, y así podarse muchas inconsistencias.

10.3.3 El problema de las N-Reinas

Este problema consiste en colocar N reinas en un tablero de ajedrez de dimensión NxN, de forma que ninguna reina esté amenazada. De esta forma no puede haber dos reinas en la misma fila, misma columna, o misma diagonal. Si asociamos cada columna a una variable y su valor representa la fila donde se coloca una reina, el problema puede ser formulado como:

- Variables: $\{x_i\}$, $i = 1..N$.
- Dominio = $\{1, 2, 3, \ldots, N\}$, para todas las variables.
- Restricciones: $(\forall x_i, x_j, i \neq j)$:
  - $x_i \neq x_j$, No en la misma fila
  - $x_i - x_j \neq i - j$, No en la misma diagonal SE
  - $x_j - x_i \neq i - j$, No en la misma diagonal SO

Resultando para el caso particular de N=4 (véase la Figura 10.3):

- $\mid x_1 - x_2 \mid \neq 1$;
- $\mid x_1 - x_3 \mid \neq 2$;
- $\mid x_1 - x_4 \mid \neq 3$;

- $\mid x_2 - x_3 \mid \neq 1$;
- $\mid x_2 - x_4 \mid \neq 2$;
- $\mid x_3 - x_4 \mid \neq 1$;
- $\neq (x_1, x_2, x_3, x_4)$;

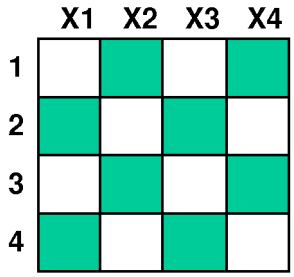


Figura 10.3: Tablero para modelado del problema de las 4-Reinas

En los ejercicios propuestos al final del capítulo se muestran otros ejemplos sobre la idoneidad de una buena especificación del CSP, tanto respecto a la simplicidad de la especificación, como con respecto a la eficiencia en el proceso de búsqueda.

10.4 Técnicas CSP

Las técnicas más usuales que se llevan a cabo para manejar un CSP se pueden agrupar en tres tipos: Búsqueda sistemática, técnicas inferenciales y técnicas híbridas.

10.4.1 Métodos de Búsqueda

Los métodos de búsqueda se centran en explorar el espacio de estados del problema. Estos métodos pueden ser *completos*, explorando todo el espacio de estados en busca de una solución, o *incompletos* si solamente exploran una parte del espacio de estados. Los métodos que exploran todo el espacio de búsqueda garantizan encontrar una solución, si existe, o demuestran que el problema no es resoluble. La desventaja de estos algoritmos es que son muy costosos. Los dos métodos completos más usuales son:

- **Generar y Testear (GT):** Este método genera las posibles tuplas de instanciación de todas las variables de forma sistemática y después testea sucesivamente sobre cada instanciación si se satisfacen todas las restricciones del problema. La primera combinación que satisfaga todas las restricciones, será la solución al problema. Mediante este procedimiento, el número de combinaciones generado por este método es el Producto Cartesiano de la cardinalidad

de los dominios de las variables: $\prod_{i=1,n} d_i$. Esto es un inconveniente, ya que se realizan muchas instanciaciones erróneas de valores a variables que después son rechazadas en la fase de testeo. Por ejemplo, para el caso del problema de las 4-Reinas, se generarían $4^4 = 256$ tuplas a testear. El proceso de generación requeriría $256 * 4 = 1024$ asignaciones de variable.

- **Backtracking Cronológico (BT):** Este método realiza una exploración en profundidad del espacio de búsqueda, instanciando sucesivamente las variables y comprobando ante cada nueva instanciación si las instanciaciones parciales ya realizadas son localmente consistentes. Si es así, sigue con la instanciación de una nueva variable. En caso de conflicto, intenta asignar un nuevo valor a la última variable instanciada, si es posible, y en caso contrario retrocede a la variable asignada inmediatamente anterior.

El Algoritmo 10.1 describe el típico proceso de BT aplicado a la resolución de CSP, donde $V[n]$ representa el vector de asignaciones a las variables $(x_1, x_2, \ldots, x_n)$ del problema, ordenadas según un determinado criterio. La función $Comprobar(K, V[n])$ comprueba la validez de las k instanciaciones ya realizadas $\{V[1], V[2], \ldots, V[k]\}$, es decir, si se satisfacen las restricciones locales existentes entre las variables $(x_1, x_2, \ldots, x_k)$, como se indica en la Expresion 10.1.

$$Comprobar(k, V[n]) = Tsii : \forall i(1 \leq k) : (V[k], V[i]) \in c_{ki} \tag{10.1}$$

Si el algoritmo finaliza con éxito, el vector $V[n]$ representa la tupla de valores solución. La Figura 10.4 muestra una sencilla aplicación del algoritmo, donde se indican los puntos de backtracking y la solución encontrada. Los algoritmos tipo BT realizan solo una comprobación *hacia atrás* en cada instanciación, por lo que pertenece al tipo de algoritmos *Look-Backward* (Sección 10.4.3.1).

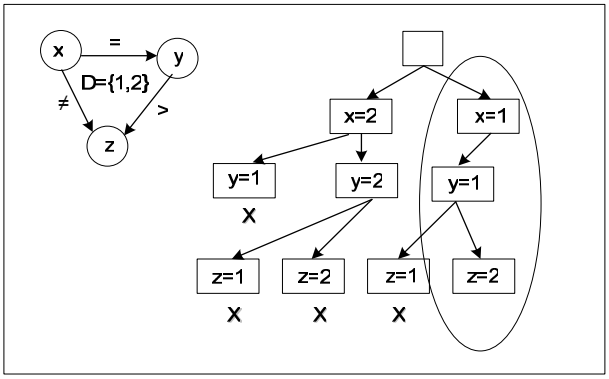


Figura 10.4: Aplicación del Backtracking Cronológico.

La Figura 10.5 muestra la aplicación del algoritmo BT al problema de las 4-reinas, con una ordenación lexicográfica de variables y valores. Los nodos marcados con X representan que la asignación realizada no es consistente con las asignaciones previas,

Algoritmo 10.1 Algoritmo de Backtracking Cronológico.

```
procedimiento Backtracking(k, V[n])    ; Llamada inicial: Backtracking(1, V[n])
inicio
V[k] = Selección(d_k)    ; Selecciona un valor de d_k para asignar a x_k
si Comprobar(k, V[n]) entonces
   si k = n entonces
      devolver V[n]    ; Es una solución
   si no
      Backtraking(k + 1, V[n])
   fin si
si no
   si quedan_valores(d_k) entonces
      Backtraking(k, V[n])
   si no
      si k = 1 entonces
         devolver ∅    ; Fallo
      si no
         Backtraking(k − 1, V[n])
      fin si
   fin si
fin si
fin Backtracking
```

y por tanto son puntos de backtracking. Como cabía esperar, BT no realiza tanta búsqueda como GT, al detectar cada inconsistencia en cuanto aparece. Pero para este simple problema, aún requiere generar 26 asignaciones.

El backtracking cronológico es un algoritmo muy simple pero también es muy ineficiente. El problema es que tiene una visión local del problema, de forma que solo comprueba las restricciones entre las variables ya instanciadas, e ignora la relación entre dichas variables $(x_1, x_2, \ldots, x_k)$ y las que quedan por instanciar $(x_{k+1}, \ldots, x_n)$. La mejora del algoritmo incluye dos modificaciones fundamentales:

- Mejorar la función *Comprobar*, tal que también compruebe la validez de la instanciación parcial efectuada respecto a las variables pendientes de asignar. Esto permitiría detectar cuanto antes si la solución parcial efectuada de las k variables forma o no parte de una solución global, evitando profundizar en una secuencia de asignaciones que no conduzca a una solución. Esto se realiza mediante un proceso de propagación de las instanciaciones parciales ya realizadas al resto de la red tal y como veremos en la sección 10.4.3.

- Establecer un orden de instanciación de las variables (orden del vector $V[n]$) o establecer un criterio para la *selección* de los valores de los dominios correspondientes ($Selección(d_k)$). Estos métodos para la ordenación de variables y valores se verán en la sección 10.5.

10.4.2 Técnicas de Inferencia

Los procesos inferenciales en un CSP tienen como objetivo deducir nuevas restricciones, derivadas de las explícitamente conocidas sobre el problema. Concretamente,

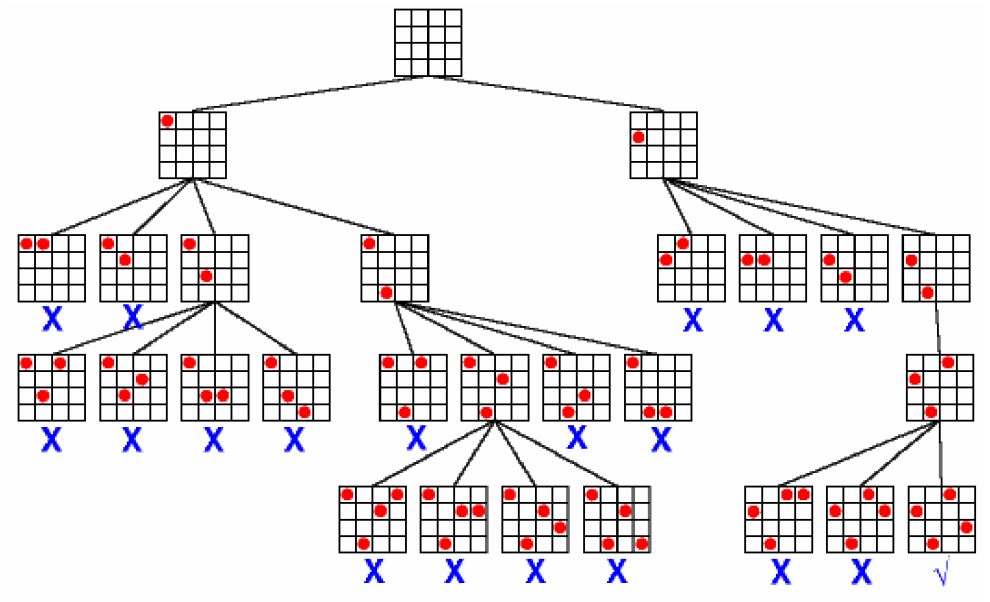


Figura 10.5: Backtracking Cronológico aplicado al problema de las 4-Reinas.

estas técnicas borran valores inconsistentes de los dominios de las variables o inducen restricciones implícitas entre las variables, obteniendo un nuevo CSP, equivalente al inicial, donde se han hecho explícitas las nuevas restricciones implícitamente contenidas en el primero. La obtención de estas nuevas restricciones permite:

1. Obtener respuestas a preguntas sobre el problema, relativas a restricciones implícitamente existentes entre las variables o sobre sus dominios.

2. Acotar el espacio de soluciones, al haberse eliminado valores inconsistentes, haciendo más eficientes los procesos de búsqueda.

Los algoritmos de búsqueda sistemática para la resolución de CSP vistos en la Sección 10.4.1 tienen como base la búsqueda basada en backtracking. Sin embargo, esta búsqueda sufre con frecuencia una explosión combinatoria en el espacio de búsqueda, y por lo tanto no es por sí solo un método suficientemente eficiente para resolver CSP. Una de las principales dificultades es la aparición de inconsistencias locales que van apareciendo continuamente [Mackworth, 1977]. Las inconsistencias locales son valores individuales o combinación de valores de las variables que no pueden participar en la solución. Por ejemplo, si el valor a de la variable x es incompatible con todos los valores de una variable y pendiente de asignación, ligada a x mediante una restricción, entonces a es inconsistente y no formará parte de ninguna solución del problema. Por lo tanto si forzamos alguna propiedad de *consistencia local* podemos borrar todos los valores que son inconsistentes con respecto a dicha propiedad. Esto no significa

que todos los valores que no puedan participar en una solución sean borrados. Puede haber valores que son consistentes con respecto a un cierto nivel de consistencia local pero son inconsistentes con respecto a cualquier otro nivel mayor de consistencia local. Así, *consistencia global* asegura que todos los valores de los dominios de las variables que no pueden participar en una solución son eliminados.

Las restricciones explícitas en un CSP, que generalmente coinciden con las que se conocen explícitamente del problema a resolver, generan cuando se combinan, restricciones implícitas que pueden causar inconsistencias locales. Si un algoritmo de búsqueda no almacena las restricciones implícitas, repetidamente redescubrirá la inconsistencia local causada por ellas y malgastará esfuerzo de búsqueda tratando repetidamente de intentar instanciaciones que ya han sido probadas. Veamos el siguiente ejemplo.

Ejemplo 10.4. *Tenemos un problema con tres variables x,y,z, con los dominios* $\{0,1\}, \{2,3\}$ *y* $\{1,2\}$ *respectivamente. Hay dos restricciones en el problema:* $y < z$, $x \neq y$. *Si asumimos que la búsqueda mediante backtracking trata de instanciar las variables en el orden x, y, z, entonces probará todas las posibles* 2^3 *combinaciones de valores para las variables antes de descubrir que no existe solución alguna. Si miramos la restricción entre la variable y y la variable z podremos ver que no hay ninguna combinación de valores para las dos variables que satisfagan la restricción. Si el algoritmo pudiera identificar esta inconsistencia local antes, se evitaría un gran esfuerzo de búsqueda.*

Los procesos inferenciales están ligados al nivel de consistencia. Un proceso inferencial completo deduciría toda la información contenida en el CSP y va ligado a un nivel de consistencia global. Sin embargo, tiene en general un coste exponencial. Los procesos inferenciales incompletos deducen solo parte de la información contenida en el CSP y van ligados a niveles de consistencia local. En general, su coste es de tipo polinómico, por lo que resultan más interesantes y aplicables. Veamos a continuación distintos niveles de consistencia en un CSP.

10.4.2.1 Consistencia de Nodo (1-consistencia)

La consistencia local más simple de todas es la *consistencia de nodo* o *nodo-consistencia*. Una variable x_i es nodo-consistente si y solo si todos los valores de su dominio D_i son consistentes con las restricciones unarias sobre la variable. Un CSP es nodo-consistente si y solo si todas sus variables son nodo-consistentes:

$$\forall x_i \in X, \forall c_i \in C, \exists a \in D_i : (a) \in c_i$$

Por ejemplo, en la Figura 10.6 consideramos una variable x en un problema con dominio $[2, 15]$ y la restricción unaria $x \leq 7$. La consistencia de nodo eliminará el intervalo $[8, 15]$ del dominio de x. El algoritmo que garantiza un CSP nodo-consistente es trivial (elimina los valores inconsistentes de los dominios de las variables respecto a sus restricciones unarias) y tiene un coste lineal respecto al tamaño del problema, habitualmente asociado al número de variables n.

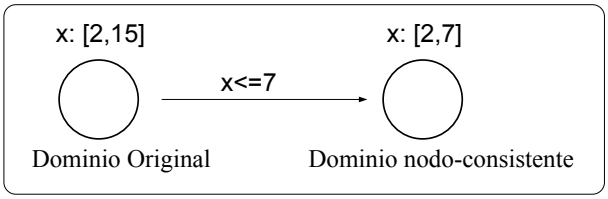


Figura 10.6: Consistencia de nodo.

10.4.2.2 Consistencia de Arco (2-consistencia)

Dos variables (x_i, x_j) son arco consistentes si para cada valor a en D_i hay al menos un valor b en D_j tal que se satisface la restricción existente entre x_i, x_j. Debe tenerse en cuenta que una restricción c_{ij} implica una restricción simétrica c'_{ji}. Un CSP es *arco-consistente* si cada par de variables del CSP es arco-consistente:

$$\forall c_{ij} \in C, \forall a \in D_i, \exists b \in D_j : (a, b) \in c_{ij}$$

Cualquier valor en el dominio D_i de la variable x_i que no es arco-consistente es eliminado de D_i, ya que no puede formar parte de solución alguna. Dicho de otra manera, una solución parcial de una variable del problema puede ser extendida a una solución parcial con cualquier otra variable. Los algoritmos de arco-consistencia recorren todos los arcos de la red, por lo que tienen un coste cuadrático respecto al número de variables n. El Algoritmo 10.2 describe un algoritmo eficiente que garantiza la arco consistencia de un CSP [Mackworth, 1977], acotando los dominios de las variables para que se cumpla dicha propiedad.

En el ejemplo de la Figura 10.7, podemos observar que el problema es arco consistente, ya que para cada valor $a \in [3, 6]$ hay al menos un valor $b \in [8, 10]$ de manera que se satisface la restricción $x_i < x_j$. Sin embargo, si la restricción fuese $x_i = x_j$ no sería arco-consistente.

Los procesos de k-consistencia suelen aplicarse antes de los procesos de búsqueda de la solución a fin de acotar los dominios y restricciones de la red, restringiendo con ello el espacio de búsqueda. Por ejemplo, en la Figura 10.8, que corresponde a un caso particular del coloreado de mapas, podemos ver que el proceso de arco consistencia elimina suficientes valores de los dominios de las variables tal que la solución es directamente obtenida sobre los valores restantes de los dominios. También puede verse el ejercicio resuelto 1 al final del capítulo, donde se muestra otro ejemplo de arco-consistencia.

Un caso particular de arco-consistencia es la *arco-consistencia dirigida*. Ello solo exige que para cualquier valor $a \in D_i$ de cualquier variable x_i, y para todos los arcos *dirigidos* c_{ij}, exista un valor b en el dominio D_j que satisfaga la restricción $(x_i c_{ij} x_j)$.

10.4.2.3 Consistencia de caminos (3-consistencia)

La consistencia de caminos [Montanari, 1974] o *senda-consistencia* (en inglés, Path-Consistency), es un nivel más alto de consistencia local que la arco-consistencia.

Algoritmo 10.2 Algoritmo para garantizar la Arco-consistencia

```
Procedimiento AC(X, n, D, C)    ; Devuelve el CSP arco-consistente
inicio
Q ← C    ; queue: cola de arcos, inicialmente los arcos en csp.
mientras Q ≠ ∅ hacer
   c_ij ← REMOVE_FIRST(Q)    ; c_ij restricción entre x_i, x_j
   si Revisar(c_ij) entonces
      para todo x_k ∈ NEIGHBORS[x_i] hacer
         añadir c_ki a Q
      fin para
   fin si
fin mientras
fin AC;

Función Revisar(c_ij)    ; Retorna T si y solo si eliminamos valores de D_i
inicio
removed ← false;
para todo x ∈ DOMAIN[x_i] hacer
   si ningún_valor y ∈ DOMAIN[x_j] satisface c_ij entonces
      eliminar x de DOMAIN[x_i] ;
      removed ← true
   fin si;
fin para
devolver removed
fin Revisar;
```

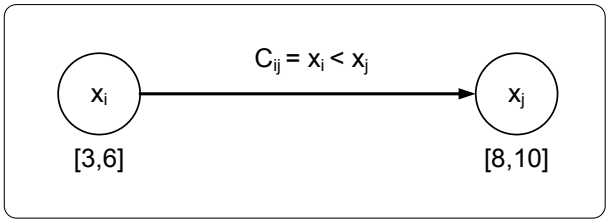


Figura 10.7: Arco-consistencia.

La consistencia de caminos requiere que, para cada asignación de dos variables (x_i, a), (x_j, b) consistente con la restricción directa que exista entre ellas c_{ij}, exista también un valor para cada variable a lo largo del camino entre ellas de forma que todas las restricciones a lo largo del camino se satisfagan (véase la Figura 10.9).

Montanari demostró que un CSP satisface la consistencia de caminos si y solo si todos los caminos de longitud dos cumplen la consistencia de caminos. En otras palabras, si y solo si cualquier solución local entre dos variables, es decir, consistente con la restricción entre ellas, es extensible a cualquier tercera variable:

$$\forall (a,b) \in c_{ij}, \forall x_k \in X, \exists c \in D_k, (a,c) \in c_{ik} \wedge (b,c) \in c_{jk}$$

En base a esta propiedad, el algoritmo PC [Mackworth, 1977; Montanari, 1974].(véase el Algoritmo 10.3) obtiene un CSP senda consistente para CSP binarios. Más conc-

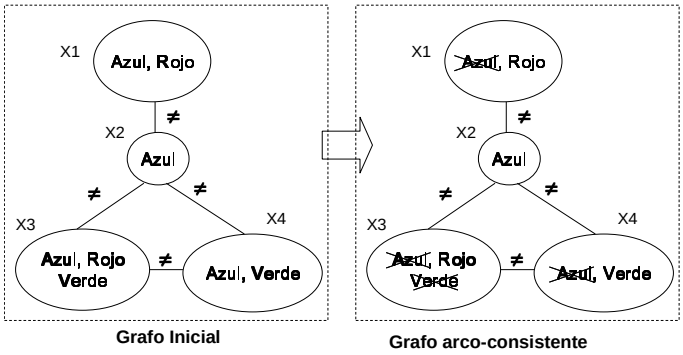


Figura 10.8: Ejemplo de proceso de Arco-Consistencia.

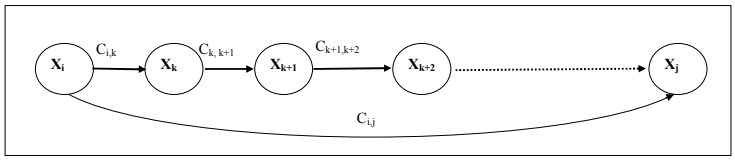


Figura 10.9: Senda Consistencia.

retamente, el algoritmo acota las restricciones del CSP inicial, de forma que el CSP resultante cumple la consistencia de caminos.

El algoritmo PC (véase Algoritmo 10.3) recorre todas las subredes de 3 nodos (conjunto Q), forzando la consistencia de senda de cada restricción c_{ij} mediante la función Revisar. En el caso de que la restricción c_{ij} resulte modificada (en su forma extensional), se vuelven a tratar todas las subredes de 3 nodos que incluyan a x_i, x_j para comprobar si siguen localmente consistentes. De esta forma, cada subred de 3 nodos, cada senda de longitud 2, resulta localmente consistente. Por lo tanto, el CSP resultante será senda consistente. En el caso en que al modificar la restricción c_{ij} quedase vacía, $c_{ij} = \{\emptyset\}$, sería un CSP no consistente.

Un ejemplo de senda-consistencia puede verse en la Figura 10.10, donde, a partir de la red inicial de restricciones R, el algoritmo obtiene la red senda consistente R': cualquier camino entre cualquier par de nodos lo es.

Una restricción c_i es consistente si existe una solución en la cual dicha restricción se cumpla. Una restricción c_i es *mínima* si y solo si consiste únicamente de tuplas consistentes, es decir, forman parte de alguna solución. Un CSP es *mínimo* si y solo si todas sus restricciones y dominios son mínimos. Puede verse que la red R' de la Figura 10.10 es senda consistente, pero no es mínima, ya que no hay ninguna solución posible donde $B = C$.

Algoritmo 10.3 Algoritmo de Senda-Consistencia.

procedimiento $PC(X, n, D, C)$
inicio
$Q := \{(i, j, k \mid 1 = i < j = n, 1 \leq k \leq n, i \neq k, j \neq k\}$; Todas las subredes de 3 nodos
mientras $Q \neq \emptyset$ **hacer**
  Select (i, j, k) *from* Q;
  si $Revisar(i, j, k)$ **entonces**
    $Q := Q \cup \{(l, i, j), (l, j, i) \mid 1 \leq l \leq n, l \neq j, l \neq i\}$
  fin si
fin mientras
fin PC;

función $Revisar(i, j, k)$
inicio
$cambiado := false$;
para todo $(a, b) \in c_{ij}$ **hacer**
  si $\neg\exists c \in d_k / (a, c) \in c_{ik} \wedge (c, b) \in c_{kj}$ **entonces**
    $c_{ij} := c_{ij} - (a, b)$; ; Acota la restriccion c_{ij}, pero $cambiado := true$
      ; si resulta $c_{ij} = \{\emptyset\}$
  fin si ;CSP no consistente
fin para
devolver *cambiado*;
fin *Revisar*;

10.4.2.4 k-consistencia. Consistencia global

En general, se dice que una red es *k-consistente* si y solo si dada cualquier instanciación de k-1 variables, que satisfagan todas las restricciones entre ellas, existe al menos una instanciación de una variable k, tal que se satisfacen las restricciones entre las k variables [Freuder, 1982]. En otras palabras, cualquier solución en una subred de tamaño k-1, puede ser extendida a una solución en una subred de tamaño k, que contenga a la anterior. En consecuencia, cada subred de k variables es localmente consistente. Cuando una red es *i-consistente* $\forall i, i \leq k$ se dice que es *fuertemente* i-consistente.

Un CSP es *consistente* si tiene al menos una solución. Un CSP es *globalmente consistente* si es i-consistente $\forall i, i \leq n$, es decir, cualquier instanciación localmente consistente entre i variables $(1 \leq i \leq n)$, puede formar parte de una solución global. De esta manera, un CSP *globalmente consistente* contiene solamente aquellas combinaciones de valores que forman parte de al menos una solución, siendo una representación compacta y conservadora de todas las soluciones de un CSP. Es correcto en el sentido de que la red nunca admite una combinación de valores que no desemboque en una solución. Es completo en el sentido de que todas las soluciones están representadas en él. Un CSP *globalmente consistente* es un CSP mínimo. En una red de restricciones globalmente consistente, la búsqueda puede llevarse a cabo sin necesidad de backtracking [Freuder, 1982].

En general, un etiquetado *globalmente consistente* de una red requiere representar de forma explícita restricciones para todas las variables del problema, es decir, generar etiquetas *n-1*-dimensionales para un problema de talla n forzando la n-consistencia.

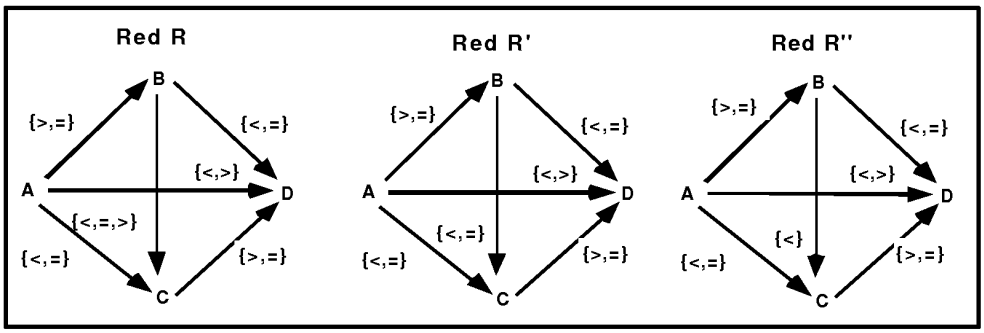


Figura 10.10: Red Inicial (R), Red Senda-Consistente (R'), Red Mínima (R").

Esta tarea tiene una complejidad exponencial en el peor caso. Para clases especiales de problemas, niveles bajos de consistencia, son equivalentes a la consistencia del CSP. Estos resultados permiten a los algoritmos polinómicos llevar a cabo etiquetados consistentes. Los podemos resumir en los siguientes:

- La Arco-consistencia es equivalente a la consistencia global cuando la red de restricciones es un árbol [Freuder, 1982].
- La consistencia de caminos es equivalente a la consistencia global cuando el CSP es convexo y binario [Dechter y otros, 1991; Van Beek, 1991], siendo un CSP convexo aquel cuyas restricciones determinan un espacio de soluciones convexo.

El problema para conocer si un CSP es consistente (tiene al menos una solución) se denomina *problema de satisfabilidad*. Este problema, así como el problema equivalente de encontrar una solución, es en general NP. Existen diferentes aproximaciones para identificar *clases tratables* de CSP, las cuales corresponden a dos líneas fundamentales:

1. Identificación por la estructura de la red de restricciones, independientemente del tipo de relaciones que contenga. En esta línea resulta interesante encontrar estructuras de búsqueda similares a árboles (véase Sección 10.5.1).
2. Identificación por el tipo de relación entre variables, debido a las especiales propiedades de dichas relaciones. Particularmente, el problema de obtener la consistencia (o encontrar soluciones) en una red binaria no disyuntiva es n^3. Asimismo, en una red binaria con las relaciones $\{<, =, >\}$, el problema de obtener la consistencia tiene un coste n^4.

10.4.3 Técnicas Híbridas

En la Sección 10.4.1 se han visto técnicas de búsqueda de soluciones en un CSP que, en general, resultan exponenciales con el tamaño del problema. Por otra parte, en la Sección 10.4.2 se han visto técnicas inferenciales que eliminan valores inconsistentes de los dominios de las variables o inducen nuevas restricciones implícitas entre

las variables restringiendo las previas, de forma que obtienen un nuevo CSP, más restringido y equivalente al inicial, que permite acotar el espacio de búsqueda.

Por ello, las técnicas inferenciales se usan como etapas de preproceso donde se detectan y se eliminan inconsistencias locales antes de empezar la búsqueda con el fin de reducir el árbol de búsqueda. Dependiendo del nivel de consistencia del preproceso, se acotará más o menos espacio de búsqueda, a costa de un mayor o menor esfuerzo computacional en el proceso inferencial previo. Por otra parte, las técnicas inferenciales pueden incluirse en el propio proceso de búsqueda, dando lugar a los *algoritmos de búsqueda híbridos* que analizamos en esta sección. La complejidad exponencial para resolver un CSP dado está principalmente relacionada con el tamaño de los dominios de sus variables.

10.4.3.1 Algoritmos Look-Backward

Los algoritmos Look-Backward tratan de explotar la información del problema para comportarse más eficientemente en las situaciones sin salida. Al igual que el backtracking cronológico, los algoritmos Look-Backward llevan a cabo la comprobación de la consistencia *hacia atrás*, es decir, entre la variable actualmente instanciada y las pasadas ya instanciadas. Veamos algunas variantes de algoritmos Look-Backward.

- **Backjumping** (BJ) [Gaschnig, 1979] es un algoritmo parecido al backtracking cronológico excepto que se comporta de una manera más inteligente cuando encuentra situaciones sin salida. En vez de retroceder a la variable anteriormente instanciada, BJ salta a la variable x_j más profunda, es decir más cerca de la variable actual, que está en conflicto con la variable actual x_i donde $j < i$. Decimos que una variable instanciada x_j está en conflicto con una variable x_i si la instanciación de x_j evita uno de los valores en x_i, debido a la restricción entre x_j y x_i. Cambiar la instanciación de x_j puede hacer posible encontrar una instanciación consistente de la variable actual.

- **Conflict-directed Backjumping** (CBJ) [Prosser, 1993] tiene un comportamiento de salto hacia atrás más sofisticado que BJ. Cada variable x_i tiene un *conjunto conflicto* formado por las variables pasadas que están en conflicto con x_i. En el momento en el que la comprobación de la consistencia entre la variable actual x_i y una variable pasada x_j falla, la variable x_j se añade al conjunto conflicto de x_i. En una situación sin salida, CBJ salta a la variable más profunda en su conjunto conflicto, por ejemplo, x_k, donde $k < i$. Al mismo tiempo se incluye el conjunto conflicto de x_i al de x_k, por lo que no se pierde ninguna información sobre conflictos. Obviamente, CBJ necesita unas estructuras de datos más complicadas para almacenar los conjuntos conflicto.

- **Learning** [Frost y Dechter, 1994] es un método que almacena las restricciones implícitas que son derivadas durante la búsqueda y las usa para podar el espacio de búsqueda. Por ejemplo, cuando se alcanza una situación sin salida en la variable x_i, entonces sabemos que la tupla de asignaciones $((x_1, a_1), \ldots, (x_{i-1}, a_{i-1})$ nos lleva a una situación sin salida. Así, nosotros podemos *aprender* que una combinación de asignaciones para las variables x_1, x_{i-1} no está permitida.

10.4.3.2 Algoritmos Look-Ahead

Como ya indicamos anteriormente, los algoritmos Look-Backward tratan de reforzar el comportamiento del backtracking mediante un comportamiento más inteligente cuando se encuentran situaciones sin salida. Sin embargo, todos ellos todavía llevan a cabo la comprobación de la consistencia solamente hacia atrás: detectan las inconsistencias en cuanto aparecen, pero ignoran las consecuencias de las instanciaciones parciales ya realizadas sobre las restricciones y dominios existentes en las futuras variables a instanciar.

Los algoritmos *Look-Ahead* hacen una comprobación *inferencial* hacia adelante en cada instanciación, integrando un proceso inferencial durante el propio proceso de búsqueda, por lo que también se denominan *técnicas híbridas*. Esto también se conoce como la *propagación* de los efectos de cada nueva instanciación al resto de la red. Ello permite: (i) acotar las restricciones y dominios de las variables futuras a instanciar, limitando el espacio de búsqueda pendiente, y (ii) encontrar las inconsistencias antes de que aparezcan, en el caso de que las instanciaciones parciales efectuadas se descubran inconsistentes con el resto de variables pendientes. En definitiva, intentan descubrir si la actual asignación localmente consistente de las k variables puede ser extendida a una solución global, provocando en caso contrario un punto de backtracking.

Forward checking (FC)

El algoritmo de look-forward más común es Forward checking. En este algoritmo, en cada etapa de la búsqueda BT, se comprueba hacia adelante la asignación de la variable actual con todos los valores de las futuras variables que están restringidas con la variable actual. Los valores de las futuras variables que son inconsistentes con la asignación actual son temporalmente eliminados de sus dominios. Si el dominio de una variable futura se queda vacío, la instanciación de la variable actual se deshace y se prueba con un nuevo valor. Si ningún valor es consistente, entonces se lleva a cabo el backtracking cronológico.

De esta manera, el algoritmo FC resulta como si en el algoritmo BT (véase el Algoritmo 10.1), la función *Comprobar* hiciera las siguientes comprobaciones:

1. $\forall i(1 \leq i < k) : (V[k], V[i]) \in c_{ki}$ (* Standard Bactracking *)
2. $\forall f(k < f \leq n), \exists V_f \in D_f : (V[k], V[f]) \in c_{kf}$ (*Forward Chaining*)

Sin embargo, el paso (1) ya no es necesario, ya que debido al proceso FC (paso 2), cada vez que una nueva variable es asignada, sabemos que todos los valores de su dominio son consistentes con los valores de las variables previamente asignadas. Por ello, no es necesario comprobar cada nueva asignación con las variables previamente asignadas (paso 1).

FC garantiza que, en cada etapa, la solución parcial actual es consistente con cada valor de cada variable futura. Además cuando se asigna un valor a una variable, solamente se comprueba hacia adelante con las futuras variables con las que están involucradas. Así mediante la comprobación hacia adelante, FC puede identificar antes

las situaciones sin salida y podar el espacio de búsqueda. El proceso de forward checking puede verse como la aplicación de un simple paso de arco-consistencia entre la variable que acaba de instanciarse y cada una de las variables que quedan por instanciar. El pseudo código de algoritmo forward checking es:

1. Seleccionar x_i.
2. Instanciar $x_i \leftarrow a_i : a_i \in D_i$.
3. Razonar hacia adelante (forward-check):
   - Eliminar de los dominios de las variables $(x_{i+1}, \ldots, x_n)$ aún no instanciadas, aquellos valores inconsistentes con respecto a la instanciación (x_i, a_i), de acuerdo al conjunto de restricciones.
4. Si quedan valores posibles en los dominios de todas las variables por instanciar, entonces:
   - Si $i < n$, incrementar i, e ir al paso (1).
   - Si $i = n$, salir con la solución.
5. Si existe una variable por instanciar, sin valores posibles en su dominio, entonces retractar los efectos de la asignación $x_i \leftarrow a_i$. Hacer:
   - Si quedan valores por intentar en D_i, ir al paso (2).
   - Si no quedan valores:
     - Si $i > 1$, decrementar i y volver al paso (2).
     - Si $i = 1$, salir sin solución.

En la Figura 10.11 se puede apreciar la aplicación del proceso FC sobre el ejemplo de CSP correspondiente al problema de las 4-reinas (descrito en la sección 10.3.3). Comprobamos la mejora respecto al algoritmo BT (Figura 10.5), ya que FC requiere generar solo 8 asignaciones. En la primera asignación $x_1 = 1$, se podan (paso 3) los dominios de las variables x_2, x_3 y x_4 de acuerdo a las restricciones que existen entre dichas variables y la variable x_1. Tras la asignación $x_2 = 3$, comprobamos (paso 5) que la variable x_4 queda sin valores en el dominio. Esto quiere decir que la instanciación localmente consistente $\{(x_1 = 1), (x_2 = 3)\}$ no puede formar parte de ninguna solución. Ello provoca un punto de backtracking sin necesidad de seguir explorando esta rama. También, comprobamos que cada asignación permite limitar los dominios de las variables pendientes de asignar, acotando de esta forma el espacio de búsqueda pendiente. Por ejemplo, la asignación $x_1 = 2$ permite limitar los dominios iniciales $\{1, 2, 3, 4\}$ de las variables pendientes a $d_2 = \{4\}$, $d_3 = \{1, 3\}$ y $d_4 = \{1, 3, 4\}$.

Forward checking se ha combinado con algoritmos Look-Backward para generar otros algoritmos *híbridos*. Por ejemplo, *forward checking* con *conflict-directed backjumping* (FC-CBJ) [Prosser, 1993] combina el movimiento hacia delante de FC con el movimiento hacia atrás de CBJ y de esa manera tiene las ventajas de ambos algoritmos.

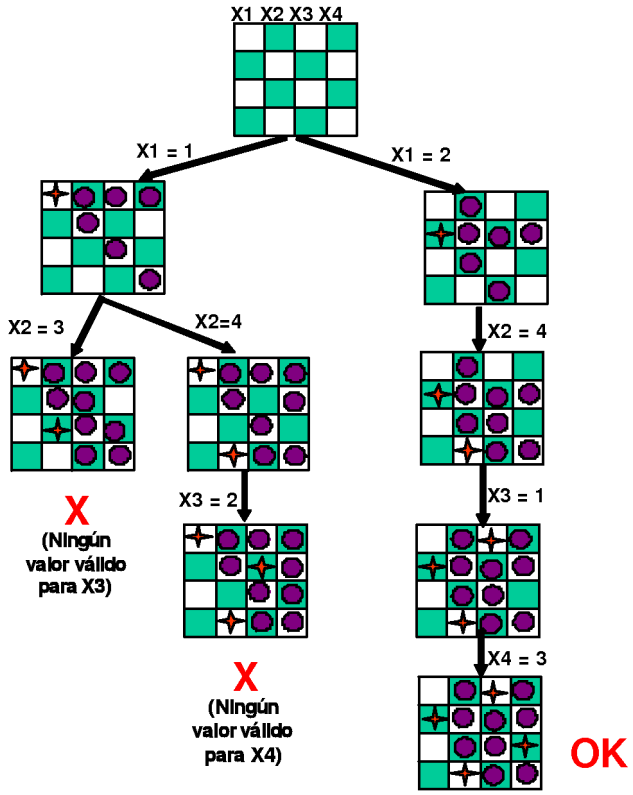


Figura 10.11: Aplicación del Forward Checking al problema de las 4-reinas.

Minimal forward checking (MFC) [Dent y Mercer, 1994] es una version de FC que retrasa llevar a cabo toda la comprobación de la consistencia de FC hasta que es absolutamente necesario. En vez de comprobar hacia adelante la asignación actual contra los valores de las variables futuras, MFC solo comprueba con los valores de cada variable futura hasta que se encuentra un valor que es consistente. Después, si el algoritmo retrocede llevaría a cabo las comprobaciones con los valores restantes aun no comprobados. Claramente, MFC siempre lleva a cabo a lo sumo el mismo número de comprobaciones que FC. Resultados experimentales han demostrado que la ganancia no supera el 10 %.

Otras versiones de FC están basadas en la idea de desarrollar un mecanismo de poda de dominios que elimina valores no solo en el nivel actual de búsqueda, sino también en cualquier otro nivel. *Extended forward checking* (EFC) [Bacchus, 2000], tiene la habilidad de podar futuros valores que son inconsistentes con asignaciones hechas antes de la asignación actual pero que no habían sido descubiertas. Otros algoritmos, como el *conflict based forward checking* (CFFC), están basados en la idea de los conflictos, al igual que los algoritmos CBJ y learning analizados en la Sección

10.4.3.1, para reforzar a FC. CFFC almacena los conjuntos conflicto y lo usa para podar los valores en los niveles pasados de búsqueda. Una diferencia con CBJ es que los conjuntos conflicto se almacenan sobre un valor y no sobre una variable, es decir, cada valor de cada variable tiene su propio conjunto conflicto. Esto permite saltar más lejos que CBJ.

Full Look- Ahead (FLA)

Como hemos analizado, el algoritmo FC comprueba únicamente las restricciones entre la variable que acaba de instanciarse con el resto de variables por instanciar. Por ello, mantiene la arco consistencia solo entre la variable asignada y el resto de variables. El algoritmo *Full Look-Ahead* (FLA) intenta mantener la arco consistencia también entre el resto de dichas variables, por lo que también es llamado *maintaining arc consistency* (MAC).

De esta manera, el algoritmo FLA resulta como si en el algoritmo BT (véase la Figura 3), la función *Comprobar* hiciera las siguientes comprobaciones (sabiendo que el paso 1 realmente ya no es necesario):

1. $\forall i(1 \leqslant i < k) : (V[k], V[i]) \in c_{ki}$ (* Standard Backtracking *)
2. $\forall f(k < f \leq n), \exists V_f \in D_f : (V[k], V[f]) \in c_{kf}$ (*Forward Checking*)
3. $\forall p(k < p \leq n), \forall q(k < q \leq n), p \neq q, \exists V_p \in D_p, \exists V_q \in D_q : (V_p, V_q) \in c_{p,q}$

El punto (3) comprueba que, tras la poda en los dominios de las variales por instanciar realizadas en FC (en el paso (2)), cada par de variables V_p, V_q que quedan por instanciar, tengan al menos un valor en sus dominios que sean consistentes con la restricción que exista entre ellas.

La ventaja del algoritmo FLA frente al FC es que detecta también los conflictos entre las variables futuras. Por ello, FLA permite podar mucho más los dominios de las variables por instanciar así como encontrar antes los puntos de backtracking debido a situaciones sin salida. Ello, evidentemente, a costa de un mayor esfuerzo en la propagación de los efectos de cada nueva asignación. Habitualmente, basta con realizar un proceso de FC frente al proceso FLA más costoso. Sin embargo, para problemas altamente restringidos el proceso FLA resulta más conveniente al podar muy fuertemente el espacio de búsqueda.

En la Figura 10.12 vemos la aplicación del proceso FLA sobre el ejemplo de la 4-reinas. Las líneas punteadas corresponden al proceso de propagación. Comprobamos la mejora respecto al algoritmo FC (véase la Figura 10.11), ya que ahora solo se requiere generar 5 asignaciones.

10.5 Heurísticas de búsqueda

Un algoritmo de búsqueda para la satisfacción de restricciones requiere establecer el orden en el cual se van a estudiar las variables, así como el orden en el que se van a

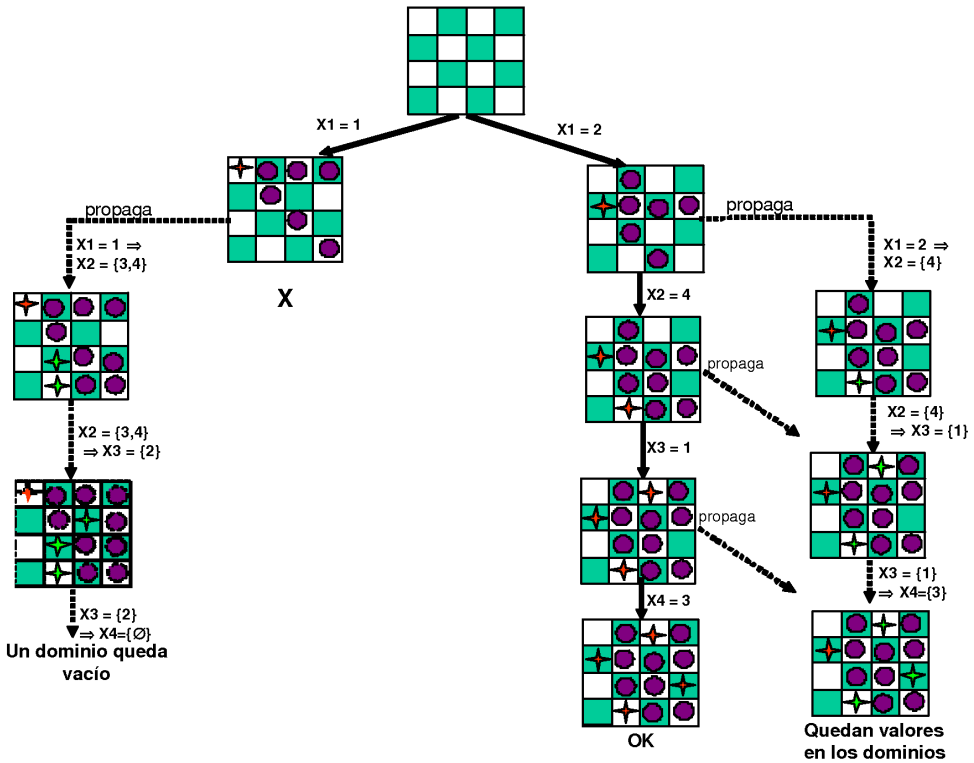


Figura 10.12: Aplicación de Look-Ahead al problema de las 4-reinas.

instanciar los valores de los dominios de cada una de las variables. Seleccionar el orden correcto de las variables y de los valores puede mejorar notablemente la eficiencia de resolución. En el método general de búsqueda mediante Backtracking descrito en la Sección 10.4.1 (véase el Algotimo 10.1), la ordenación de variables se corresponde a la ordenación que se realice del vector $V\,[n]$, mientras que los criterios para la selección de valores se incluirían en la función *Selección* incluida en el Algoritmo 10.1. También puede resultar importante una ordenación adecuada de las restricciones del problema cuando se aplique la función *Comprobar* [Salido y Barber, 2006], aunque se ha realizado mucho menos esfuerzo en el estudio de tales heurísticas. Veamos las más importantes heurísticas de ordenación de variables y de ordenación de valores.

10.5.1 Heurísticas de Ordenación de Variables

El orden en el cual las variables son asignadas durante la búsqueda puede tener un impacto significativo en el tamaño del espacio de búsqueda explorado. Esto puede ser contrastado tanto empírica como analíticamente. Generalmente las heurísticas de ordenación de variables tratan de seleccionar lo antes posible las variables que más

restringen a las demás. La intuición es tratar de asignar lo antes posible las variables más restringidas y de esa manera identificar las situaciones sin salida lo antes posible y así reducir el número de vueltas atrás. La ordenación de variables puede ser estática o dinámica.

10.5.1.1 Ordenación de Variables Estática

Las heurísticas de *ordenación de variables estática* generan un orden fijo de las variables antes de iniciar la búsqueda, basado en información global derivada de la estructura de la red de restricciones original que representa el CSP. Veamos primero unas ideas iniciales. Una *red de restricciones ordenada* es una red de restricciones cuyos nodos han sido ordenados linealmente. En la Figura 10.13, podemos observar los diferentes órdenes lineales (de arriba a abajo) posibles para una red inicial de tres variables: (a,b,c), (a,c,b), (b,a,c), etc. El *ancho* de una variable en una red de restricciones ordenada es el número de arcos que restringen esa variable con las variables superiores, denominadas *padres*, en el orden lineal de las variables establecido. El ancho de cada red de restricciones ordenada es el máximo ancho de todas sus variables. De esta forma, el *ancho de una red de restricciones* es el mínimo ancho de todas sus posibles ordenaciones lineales. Así, el ancho de la red de restricciones inicial de la Figura 10.13 es 1. Claramente, el ancho de cada red de restricciones depende de su estructura.

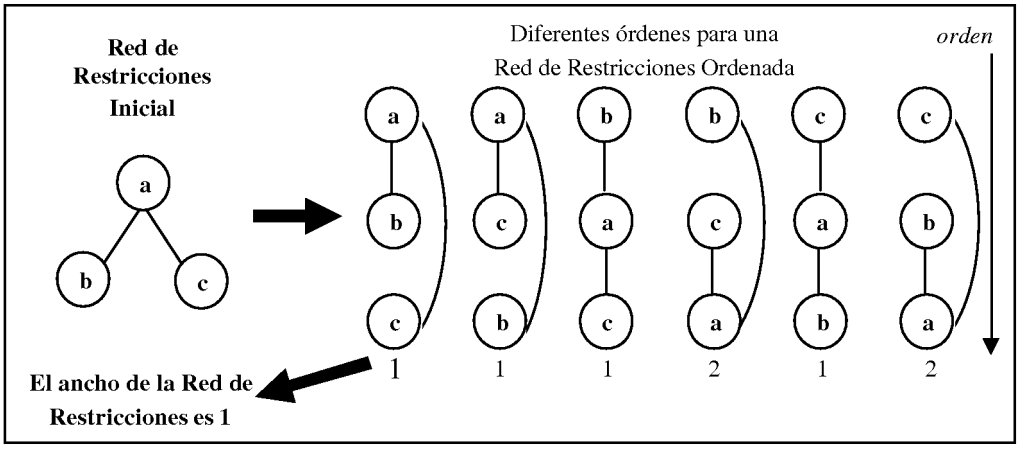


Figura 10.13: Red de restricciones ordenadas.

Si una red de restricciones es fuertemente k-consistente, siendo k mayor que el ancho de la red, entonces existe un orden de asignación de variables que no provocaría vuelta atrás en el proceso de búsqueda. Particularmente, en el caso de redes con estructura de árbol, cuyo ancho es 1, sería posible efectuar una búsqueda sin vuelta atrás si la red fuera arco consistente (2-consistencia). Más concretamente, sería suficiente con mantener la *arco-consistencia direccional* (véase la Sección 10.4.2.2), menos

costosa que la arco-consistencia. Como ejemplo, en la red de la Figura 10.13, un orden de asignación de variables correspondiente a un ordenanamiento de la red de ancho 1 no provocaría vuelta atrás. En un caso general, interesa encontrar el mejor orden para la asignación de variables. Entre las heurísticas más utilizadas podemos mencionar:

- **Minimum Width** (MW). Esta heurística se basa en que la ordenación de variables corresponda a la ordenación lineal de la red que dé lugar a su menor anchura (ancho de la red). Mediante este orden, se persigue que cuando se asigne una variable, sus *padres* estén ya asignados, de forma que las situaciones sin salida puedan identificarse antes y se reduzca el número de vueltas atrás.

  Ejemplo 10.5. *La red de la Figura 10.14 permite 7! (es decir, 5.040) ordenamientos lineales de las variables. El ancho de esta red es 1, que corresponde, entre otras, a la ordenación lineal de las variables indicada en la Figura 10.14. En dicha ordenación, todas las variables tienen un orden 1, por lo que la red inicial tiene un ancho igual a 1. La ordenación de las variables, en el orden lineal de la red, de arriba abajo, resulta* (a, b, c, f, g, d, e*). Este sería el orden de selección de variables obtenido según el criterio MW. Claramente, al tener la red inicial la estructura de un árbol, su mejor orden corresponde a asignar cuanto antes la variable* a*, luego la variable* b *o* c*, y finalmente,* d *o* e*, o bien* f *o* g*.*

  El algoritmo que obtiene el orden de las variables según el método MW selecciona, sobre la red original, la variable con el mínimo número de variables conectadas y la coloca al final del orden. Dicha variable y todos sus arcos adyacentes se eliminan de la red original. Posteriormente, se selecciona la siguiente variable, y el proceso continúa recursivamente.

  Ejemplo 10.6. *En la Figura 10.15 se muestra la aplicación del método MW a una red de ancho 3. Inicialmente, se selecciona* G *como la variable menos conectada y se elimina dicha variable y sus relaciones de la red. Posteriormente, se seleccionan* $\{F, E\}$*, después* D*,* C*,* B *y, finalmente* A*. Podríamos comprobar que este orden da lugar, entre otros, al mínimo ancho de la red:*

Orden de Selección:	$A \rightarrow$	$B \rightarrow$	$C \rightarrow$	$D \rightarrow$	$E \rightarrow$	$F \rightarrow$	G
Ancho de la Variable:	*0*	*1*	*2*	*3*	*3*	*3*	*3*

  Luego el orden de selección de variables según MW resultaría (A, B, C, D, E, F, G)*. Alternativamente, si se hubiera optado por otra ordenación* (A, B, C, D, E, G, F)*, no hubiera dado lugar al mínimo orden de la red. El ancho de las variables sería (0,1,2,3,3,2,4) respectivamente, por lo que daría lugar a un ancho 4.*

- **Maximum Degree** (MD). Esta heurística ordena las variables en un orden decreciente de su grado en la red de restricciones. El *grado* de una variable se define como el número de variables con las que está conectada. De esta forma, esta heurística selecciona primero las variables de mayor grado, es decir, las

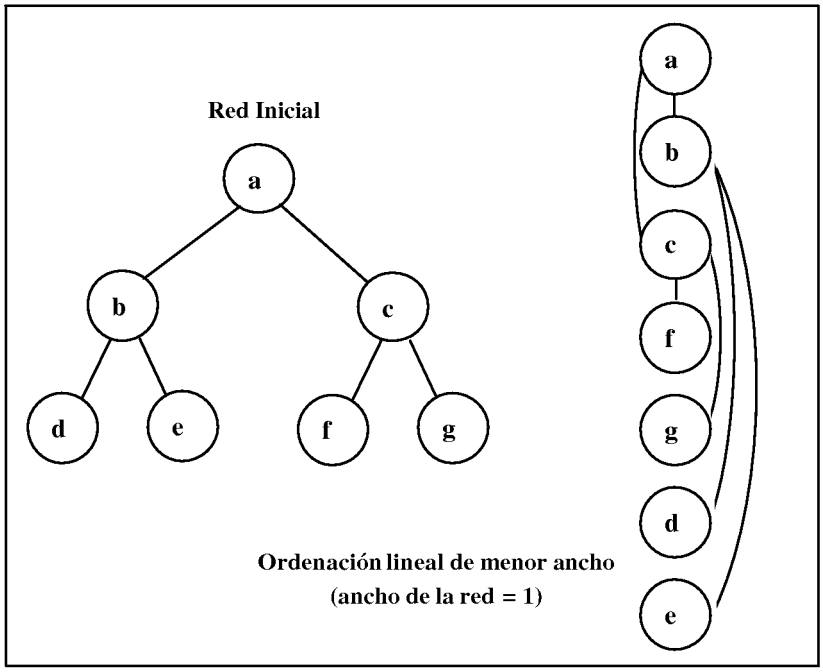


Figura 10.14: Ordenación de una red de mínimo ancho.

más conectadas. En la red de la Figura 10.15, la ordenación de selección sería $(D, A, \{B, C, E, F\}, G)$. Esta heurística intenta seguir el orden de variables de anchura mínima (MW), con un menor coste computacional para el cálculo de dicho orden, aunque no lo garantiza.

- **Mínimum Domain Variable** (MDV). Esta heurística selecciona las variables de acuerdo a la menor cardinalidad de su dominio. Las variables con dominio más pequeño son elegidas antes. Sin embargo, respecto a la aplicación de esta heurística, es preferible su versión dinámica (*minimum remaining values*).

Diversos resultados experimentales sobre heurísticas de ordenación de variables estáticas utilizando CSP generados aleatoriamente concluyen que todas ellas son peores que la heurística *minimum remaining values* (MRV), que es una heurística de ordenación de variables dinámica que veremos a continuación.

10.5.1.2 Ordenación de Variables Dinámica

El problema de los algoritmos de ordenación de variables estáticos es que no tienen en cuenta los cambios en los dominios o relaciones de las variables causados por la propagación de las restricciones durante la búsqueda. Por ello, estas heurísticas

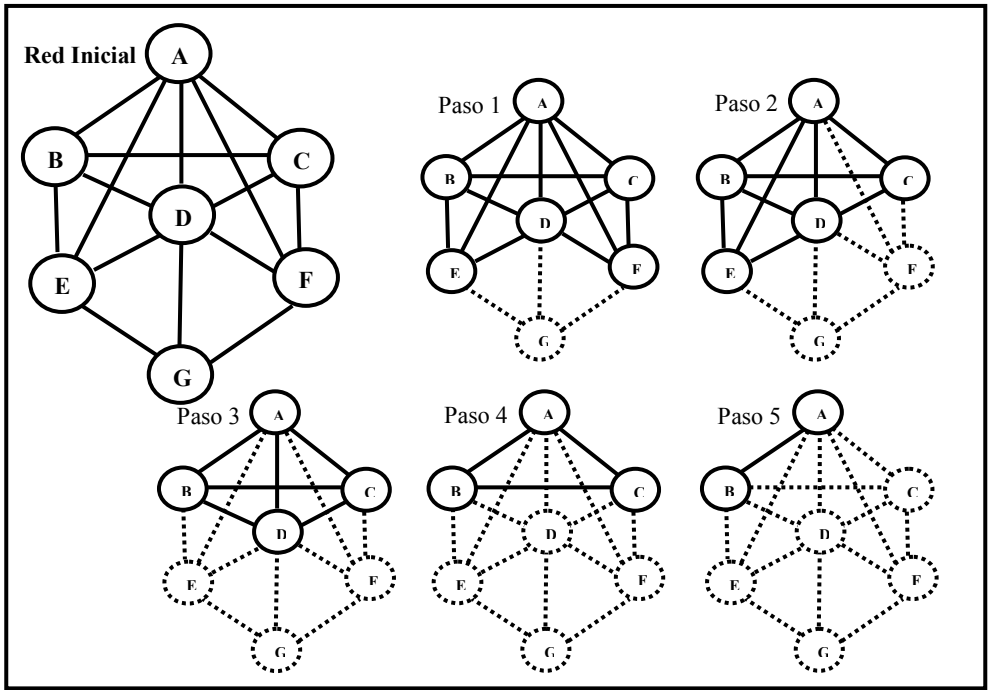


Figura 10.15: Aplicación de la heurística Minimum Width.

generalmente se utilizan en algoritmos de comprobación hacia atrás (o look-backward) donde no se lleva a cabo la propagación de las instanciaciones que se van realizando.

Las heurísticas de *ordenación de variables dinámica* pueden cambiar el orden de selección de las variables de forma dinámica durante el proceso de búsqueda, cada vez que requiere la instanciación de una variable. La ordenación se basa en las instanciaciones ya realizadas y en el estado de la red en cada momento de la búsqueda.

Se han propuesto varias heurísticas de ordenación de variables dinámicas. Las más comunes se basan en el principio de *primer fallo* que sugiere que "*para tener éxito deberíamos intentar primero donde sea más probable que falle*". De esta manera las situaciones sin salida pueden identificarse antes y además se ahorra espacio de búsqueda. De acuerdo a este principio, en cada paso, se selecciona la variable más restringida. Entre las más utilizadas podemos citar:

- **Minimum Remaining Values** (MRV). Esta heurística selecciona, en cada paso, la variable con el dominio de instanciación más pequeño. Esta heurística se basa en la intuición de que si una variable tiene pocos valores de elección en su dominio, entonces es más difícil encontrar un valor consistente. Cuando se utiliza MRV junto con algoritmos de comprobación hacia atrás (Algoritmos look-Backward descritos en 10.4.3.1), equivale a una heurística estática que ordena las variables de forma ascendente con la talla del dominio antes de llevar a

cabo la búsqueda, y resulta igual por tanto a la heurística MDV. Cuando MRV se utiliza en conjunción con algoritmos look-ahead (10.4.3.2), la ordenación se vuelve dinámica. En estos casos, cada nueva instanciación de variable es propagada al resto de la red y puede acotar los dominios de las variables que quedan por instanciar. Así, en cada paso de la búsqueda, la próxima variable a asignar es la variable con el dominio más pequeño.

- **Maximun Cardinality** (MC). Esta heurística selecciona la primera variable arbitrariamente y después en cada paso, selecciona la variable que está relacionada con el mayor número de variables ya instanciadas. La intuición es que resulta la más restringida, al estar relacionada con el mayor número de variables ya instanciadas. Esta heurística resulta similar a la heurística MD previamente definida, pero para el caso dinámico. Como una variación de la misma, podría seleccionarse, dinámicamente en cada paso, la variable de máximo grado, sin contabilizar las variables ya instanciadas.

10.5.2 Ordenación de Valores. Tipos

En comparación con la ordenación de variables, se ha realizado menos trabajo sobre heurísticas para la ordenación de valores. Estas heurísticas tienen como objetivo seleccionar el valor más prometedor para cada variable en su dominio de instanciación. Se aplicarían, por ejemplo, en el paso 2 del algoritmo Forward-Checking descrito en 10.4.3.2. La idea básica es seleccionar el valor de la variable actual que más probabilidad tenga de llevarnos a una solución, es decir, identificar la rama del árbol de búsqueda que sea más probable que obtenga una solución.

La mayoría de las heurísticas propuestas tratan de seleccionar el valor menos restringido de la variable actual, es decir, el valor que menos reduce el número de valores útiles para las futuras variables, o alternativamente, el que deja los dominios mayores. Esto sigue la intuición de que un subproblema es más probable que tenga solución cuantos más valores tengan las variables que quedan por instanciar en sus dominios. Entre las más utilizadas podemos mencionar:

- **min-conflicts**. Es una de las heurísticas de ordenación de valores más conocida. Esta heurística ordena los valores de acuerdo a los conflictos que generan con las variables aún no instanciadas. Esta heurística asocia a cada valor a de la variable actual, el número total de valores en los dominios de las futuras variables adyacentes con la actual que son incompatibles con a. El valor seleccionado es el asociado a la suma más baja.

- **max-domain-size**. Alternativamente a la anterior, esta heurística selecciona el valor de la variable actual que deja el máximo dominio en las variables futuras.

- **weighted-max-domain-size**. Esta heurística especifica una manera de romper empates en el método anterior, en el caso de que existan varios valores de la variable actual que dejen el mismo máximo dominio en las variables futuras. Entonces, se basa en el número de futuras variables que tienen la mayor talla de

dominio. Por ejemplo, si un valor a_i deja cinco variables futuras con dominios de seis elementos (y el resto con dominios mayores de seis), y otro valor a_j deja siete variables futuras también con dominios de seis elementos, en este caso se selecciona el valor a_i.

- **point-domain-size**. Esta heurística asigna un peso (unidades) a cada valor de la variable actual dependiendo del número de variables futuras que se quedan con ciertas tallas de dominios. Por ejemplo, para cada variable futura que se queda con un dominio de talla 1 debido al valor a_i, se añaden 8 unidades al peso de a_i, para cada variable futura que se queda con un dominio de talla 2 se añaden 7 unidades, etc. De esta manera se selecciona el valor con el menor peso.

- **Supervivencia**. Esta heurística propone una variación de la idea anterior de la heurística *min-conflicts*. De acuerdo a dicha heurística, cuando se cuenta el número de valores incompatibles en el dominio de una futura variable x_k, debido a la elección del valor a_i para x_i, éste número se divide por la talla del dominio de x_k. Esto da el porcentaje de los valores útiles que pierde x_k debido al valor a_i que actualmente estamos examinando. De nuevo, los porcentajes se añaden para todas las variables futuras con las que está relacionada x_i y se selecciona el valor más bajo que se obtiene en todas las sumas.

- **max-promise**. En esta heurística, para cada valor a_i de la variable x_i se cuenta el número de valores que hay compatibles con a_i en cada variable adyacente futura, y se toma el producto de las cantidades contadas. Este producto se llama *promesa del valor*. De esta manera se selecciona el valor con la máxima promesa. Al usar el producto en vez de la suma del número de valores compatibles, la heurística *max-promise* trata de seleccionar el valor que deja un mayor número de soluciones posibles después de que este valor se haya asignado a la variable actual. De esta manera, la promesa de cada valor representa una cota superior del número de soluciones diferentes que pueden existir después de que el valor se asigne a la variable.

10.6 Extensiones de CSP

En esta Sección se presentan algunas de las extensiones más relevantes a la metodología CSP, orientadas a tipologías de problemas más concretas o que plantean requerimientos adicionales.

10.6.1 CSP no binarios

Gran parte de las técnicas de satisfacción de restricciones se centran en problemas binarios, en el caso de CSP discretos, y en problemas ternarios, en el caso de CSP continuos. La razón básica de tratar con restricciones binarias y ternarias es por la simplicidad que supone comparado con las restricciones n-arias y también por el hecho

de que todo problema de satisfacción de restricciones n-arias puede ser transformado a un problema binario o a uno ternario equivalente, bajo la definición de *equivalencia de red*.

Principalmente, hay dos técnicas para convertir las restricciones no binarias a binarias en CSP discretos:

- **Codificación dual**, introducida por Dechter y Pearl, en la cual las restricciones del problema original se transforman en variables del nuevo problema y viceversa. Las variables que se generan de las restricciones originales se denominan *variables duales*, y las variables del problema original se denominan *variables originales*. El dominio de cada variable dual es el conjunto de tuplas que satisfacen la restricción original que la genera. Además, hay una restricción binaria entre dos variables duales v_c y v'_c si y solamente si las restricciones originales c y c' comparten una o más variables. Las nuevas restricciones binarias se denominan restricciones duales y prohíben pares de tuplas donde las variables compartidas tomen valores diferentes.

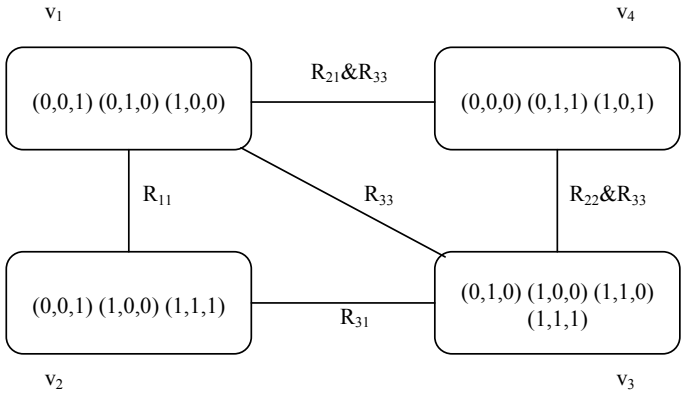


Figura 10.16: Ejemplo de codificación dual de un CSP no binario.

Si consideremos el siguiente ejemplo, tomado de [Stergiou y Walsh, 1999], con seis variables con dominios $\{0, 1\}$ y cuatro restricciones: $x_1+x_2+x_6 = 1$, $x_1-x_3+x_4 = 1$, $x_4+x_5-x_6 \geq 1$ y $x_2+x_5-x_6 = 0$. La codificación dual representa este problema con cuatro variables duales, una para cada restricción. Los dominios de estas variables duales son las tuplas que satisfacen las respectivas restricciones. Por ejemplo, la variable dual v_3 asociada a la tercera restricción, $x_4+x_5-x_6 \geq 1$, tiene como dominio $(0, 1, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1)$, ya que estas son las tuplas de valores para (x_4, x_5, x_6) que satisfacen la restricción. La codificación dual del problema se muestra en la Figura 10.16. R_{ij} es la restricción binaria sobre un par de tuplas que se satisface si y solo si el $i-$ésimo elemento de la primera tupla es igual al $j-$ésimo elemento de la segunda tupla. Entre v_3 y v_4 hay una restricción de compatibilidad para asegurar que las dos variables originales en común, x_5 y x_6 tienen los mismos valores. Esta restricción permite solamente

aquellos pares de tuplas que concuerdan con los elementos segundos y terceros, es decir $(1,0,0)$ para v_3 y $(0,0,0)$ para v_4.

- **Codificación de variables ocultas**, que procede de Peirce, quien, en 1933, probó formalmente en el campo de la filosofía lógica que las relaciones binarias y no binarias tienen el mismo poder de expresividad. Su método para representar una relación no binaria mediante una colección de restricciones binarias formó las bases del método de las variables ocultas. En la codificación de variables ocultas, el conjunto de variables está formado por todas las variables del CSP no binario original más un nuevo conjunto de variables duales que llamaremos *variables ocultas*. Al igual que la codificación dual, cada variable oculta corresponde a una restricción en el problema original. De nuevo, el dominio de cada variable oculta consta de las tuplas que satisfacen la restricción original. Para cada variable oculta v_c, hay una restricción binaria entre la variable v_c y cada una de las variables originales x_i que están involucradas en la correspondiente restricción original c. Cada restricción especifica que la tupla asignada a v_c es consistente con el valor asignado a x_i.

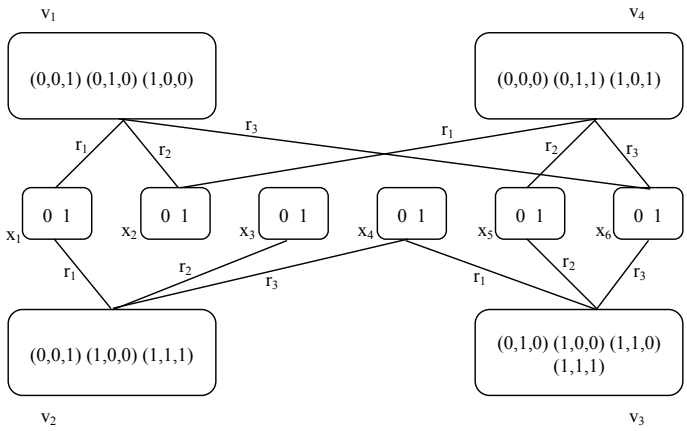


Figura 10.17: Ejemplo de codificación de variables ocultas de un CSP no binario.

Considerando de nuevo el ejemplo anterior presentado en [Stergiou y Walsh, 1999], donde el problema consta de seis variables y cuatro restricciones no binarias. En la codificación de variables ocultas hay, además de las seis variables originales con dominios $\{0,1\}$, cuatro variables duales con los mismos dominios que en la codificación dual. Por ejemplo, la variable dual asociada con la tercera restricción $x_4 + x_5 - x_6 \geq 1$, tiene como dominio $(0,1,0), (1,0,0), (1,1,0), (1,1,1)$. Además ahora hay restricciones de compatibilidad entre v_3 y x_4, entre v_3 y x_5 y entre v_3 y x_6, ya que estas son las variables que están involucradas en la tercera restricción. La codificación de variables ocultas de este problema se muestra en la Figura 10.17. La restricción binaria r_i actúa sobre una tupla y un valor, dando verdadero si y solo si el elemento $i-$ésimo de la tupla es igual al del valor. Por

ejemplo, la restricción de compatibilidad r_1 entre v_3 y x_4 es cierta si y solo si el primer elemento de la tupla asignada a v_3 es igual al valor de x_4.

En el caso de CSP numéricos o continuos, se pueden transformar las restricciones n-arias en restricciones ternarias, según el método descrito por Lottaz, de forma que se mejora la eficiencia de los algoritmos de consistencia. Esta transformación solo es posible si se permite la introducción de variables auxiliares.

La transformación de CSP n-arios en CSP binarios equivalentes permite la aplicación de las diversas técnicas conocidas para CSP binarios. Sin embargo, se pierde la legibilidad de las restricciones, originalmente adquiridas desde el análisis del problema como restricciones n-arias. Por otra parte, resultan más difíciles de aplicar heurísticas de búsqueda orientadas al dominio, al haber perdido las variables y las restricciones su equivalencia con el problema original. Por ello, se mantiene el interés de trabajar con CSP n-arios.

10.6.2 CSP distribuidos (DisCSP)

En el ámbito de los sistemas multiagente, los problemas de satisfacción de restricciones se pueden manejar mediante el uso de agentes inteligentes que se encargan de controlar subconjuntos de variables y restricciones del problema. Si todo el conocimiento sobre el problema puede ser concentrado en un único agente, este agente podrá resolver el problema mediante el uso de técnicas centralizadas de satisfacción de restricciones. Sin embargo, obtener una solución de forma centralizada es con frecuencia inadecuado o incluso imposible. Algunas razones por las que son deseables el uso de métodos distribuidos son [Faltings y Yokoo, 2005]:

- **Coste de crear una autoridad central.** Un CSP puede ser distribuido de forma natural en un conjunto de agentes concretos. En tales casos, el uso de una autoridad central para resolver el problema requiere establecer un elemento adicional que no estaba presente en la arquitectura del problema.

- **Privacidad/seguridad.** Las restricciones que maneja cada agente puede ser información estratégica que no debería ser revelada a la competencia o incluso a una autoridad central. La privacidad es más fácil de mantener en resolvedores distribuidos.

- **Robustez frente a fallos.** El fallo de un servidor central puede ser fatal. En un método distribuido, el fallo de un agente puede ser menos crítico y otros agentes podrían ser capaces de encontrar una solución al problema.

Adicionalmente, pueden existir características de jerarquía, cooperación, etc., en un problema distribuido que se puede representar de una manera más eficiente mediante una arquitectura de CSP distribuida. De esta manera, surge la necesidad de distribuir el problema en una serie de sub-problemas que cumplan con los anteriores criterios y que, además, sean más fáciles de resolver. Sin embargo, una de la complejidad fundamental en la resolución de los DisCSP es el sobrecoste que supone el

intercambio de mensajes entre los agentes. Por otra parte en los problemas distribuidos no se pueden aplicar heurísticas e inferencias de CSPs centralizados.

Un CSP Distribuido (DisCSP) es un CSP donde las variables y las restricciones son distribuidas entre múltiples agentes. Cada agente en un DisCSP tiene algunas variables y trata de determinar sus valores. Sin embargo hay restricciones inter-agentes y la asignación de valores debe satisfacer estas restricciones. La búsqueda que se lleva a cabo para resolver un DisCSP puede ser llevada a cabo por un agente central, aunque en la mayoría de los casos la búsqueda se lleva a cabo de forma distribuida a través del intercambio de mensajes entre los diferentes agentes. Existen dos aproximaciones básicas para la búsqueda de soluciones en DisCSP. La primera de ellas es la llamada *Synchronous Backtracking* (SBT). Para ello, se ordenan inicialmente todos los agentes $(A_1, A_2, \ldots, A_n)$. El primer agente A_1 obtiene una asignación consistente para sus variables y envía al siguiente agente dicha asignación, y así sucesivamente. De esta forma, cada agente espera un mensaje del agente anterior para empezar la búsqueda. Cuando un agente A_i recibe el mensaje, intenta obtener una asignación a sus variables de tal manera que se cumplan todas las restricciones, incluyendo las variables asignadas en los agentes previos. Si un agente A_i no pudiera encontrar una asignación consistente, envía un mensaje al agente anterior A_{i-1}, a fin de pedirle una asignación alternativa. La solución global se alcanza cuando el último agente A_n encuentra una asignación para sus variables. En cambio, no existe solución si el primer agente A_1 recibe un mensaje de asignación alternativa de A_2, y no puede obtenerla.

En un esquema SBT, en principio, los agentes no se ejecutan concurrentemente. Sin embargo, se han planteado esquemas que superan este inconveniente, de forma que cada agente A_i continua buscando soluciones alternativas a la previamente encontrada a fin de poder ofrecer dicha instanciación rápidamente al agente siguiente A_{i+1}, si recibe un mensaje para ello. A este esquema, también se pueden plantear las típicas alternativas de un proceso de Backtracking como el conocido *BackJumping*.

La segunda alternativa fue llevada a cabo por Yokoo y otros [Yokoo y otros, 1998] en sus algoritmos de *backtracking asíncrono*, donde el DisCSP se resuelve mediante el intercambio de mensajes de forma asíncrona. Ello permite a los agentes actuar concurrentemente sin un control global, garantizando la completitud del proceso. Para ello, se asume una ordenación de prioridades entre los agentes y cada agente es responsable de forzar todas las restricciones entre sus variables y las variables que sean propiedad de agentes de rango mayor. A partir de este planteamiento, y utilizando la ordenación de prioridades, se desarrollan dos algoritmos de búsqueda distribuida, *asynchronous backtracking* (ABT) y *asynchronous weak-commitment* (AWC). La principal diferencia entre ellos es el manejo de las situaciones sin salida (*dead-end*), es decir, cuando un agente no puede encontrar una asignación consistente para sus variables. En ABT, el agente retrocede y envía un mensaje a los agentes de prioridad superior alegando que la combinación de asignaciones de valores a las variables de rango superior no puede formar parte de una solución en sus variables, y solicitando que se cambie una asignación. Por el contrario, en AWC, un agente utiliza una técnica llamada *compromiso débil*, donde un agente renuncia a la tentativa de satisfacer sus restricciones y las delega a otros agentes levantando su propia prioridad. Mientras hace esto, un agente puede enviar instanciaciones no consistentes a otros agentes por lo que ellos

evitarán asignaciones de valores especificadas en las instanciaciones no consistentes. Resultados experimentales muestran que AWC es significativamente mejor que ABT en la obtención de soluciones en algunas instancias de DisCSP difíciles. Otras distintas aproximaciones para el manejo de DisCSP pueden ser vistas en [Faltings y Yokoo, 2005].

10.6.3 CSP Temporales y CSP Dinámicos

Un CSP temporal es un caso particular de CSP donde las variables representan una primitiva temporal (puntos de tiempo, intervalos, o duraciones temporales), estableciendose restricciones temporales entre ellas. Constituyen una tipología de CSP de gran aplicación (bases de datos, lenguaje natural, scheduling, etc.). Los modelos básicos de CSP temporales son binarios, y se basan en álgebras temporales. Un álgebra temporal define, fundamentalmente: (i) el conjunto de restricciones sobre las variables ($x_i c_{ij} x_j$), y (ii) las operaciones básicas entre las mismas. Estas operaciones son, principalmente:

- **Adición** de una nueva restricción sobre un mismo par de variables:

$$(x_i \{c_{ij}\} x_j) \oplus (x_i \{c'_{ij}\} x_j) = x_i \{c_{ij} \oplus c'_{ij}\} x_j$$

- **Combinación** (transitiva), que permite obtener la relación c_{ik} a partir de c_{ij} y c_{jk}:

$$(x_i \{c_{ij}\} x_j) \otimes (x_j \{c_{jk}\} x_k) = x_i \{c_{ij} \otimes c_{jk}\} x_k$$

- Otras operaciones son la inclusión: ($c_{ij} \subset c'_{ij}$), la negación ($\neg c_{ij}$), etc.

Los principales CSP temporales son:

- **Álgebra de Puntos** [Vilain y Kautz, 1986], que establece restricciones temporales cualitativas y disyuntivas entre puntos de tiempo: $t_i \{<, =, >\} t_j$. En la Figura 10.18 se muestran las tablas para las operaciones $\oplus$ y $\otimes$ de esta álgebra.

- **Álgebra de Intervalos** [Allen, 1983], que establece restricciones temporales cualitativas y disyuntivas entre intervalos: $I_i\{before, meets, overlaps, during, starts, finishes, equal\}\ I_j$, así como sus inversas.

- **Álgebra Métrica entre puntos** [Dechter y otros, 1991], que establece restricciones temporales métricas y disyuntivas entre puntos de tiempo: $t_j - t_i \in \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$, donde a_i, b_i representan puntos extremos de intervalos métricos.

Estas álgebras permiten especificar las típicas restricciones de scheduling respecto a la compartición no simultánea de recursos. Por ejemplo, si las tareas W_1 y W_2 deben compartir, de forma no simultánea, un recurso común, debemos especificar que su ejecución no debe solaparse en el tiempo. Es decir:

$\oplus$	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$	?
$<$	$<$	$<$	$\emptyset$	$\emptyset$	$\emptyset$	$<$	$<$
$\leq$	$<$	$\leq$	$\emptyset$	$=$	$=$	$<$	$\leq$
$>$	$\emptyset$	$\emptyset$	$>$	$>$	$\emptyset$	$>$	$>$
$\geq$	$\emptyset$	$=$	$>$	$\geq$	$=$	$>$	$\geq$
$=$	$\emptyset$	$=$	-	$=$	$=$	-	$=$
$\neq$	$<$	$<$	$>$	$>$	$\emptyset$	$\neq$	$\neq$
?	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$	?

$\otimes$	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$	?
$<$	$<$	$<$	?	?	$<$	?	?
$\leq$	$<$	$\leq$	?	?	$\leq$	?	?
$>$	?	?	$>$	$>$	$>$	?	?
$\geq$	?	?	$>$	$\geq$	$\geq$	?	?
$=$	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$	?
$\neq$	?	?	?	?	$\neq$	?	?
?	?	?	?	?	?	?	?

Figura 10.18: Tablas para las operaciones $\oplus$ y $\otimes$ en el álgebra de puntos ($\emptyset$ es la inconsistencia y ? representa $\{<=>\}$.

1. Mediante restricciones sobre los intervalos de ocurrencia de las tareas:
$$I_{w_1} \{before, after\} I_{w_2}$$

2. Mediante restricciones sobre los puntos de tiempo extremos:
$$End_{w_1} < Start_{w_2} \vee End_{w_2} < Start_{w_1}$$

3. O bien, conociendo la duración de las tareas, podemos utilizar un modelo métrico:
$$Start_{w_1} - Start_{w_2} \in \{]- \propto, -dur_{w_1}], [dur_{w_2}, \propto [\}$$

Otras álgebras temporales establecen restricciones entre duraciones, entre puntos e intervalos, y entre puntos, duraciones e intervalos, etc. Estos modelos plantean ya restricciones ternarias.

Los CSP temporales binarios se pueden representar mediante una red de restricciones temporales (Temporal Constraint Network o TCN), como un grafo dirigido, donde una restriccion entre x_i, x_j, implica una restricción simétrica entre x_j, x_i.

Consideremos el siguiente ejemplo [Dechter y otros, 1991]: "Juan va de su casa al trabajo en coche (30-40 minutos) o en tren (al menos una hora). Luis va en coche (20-30 minutos) o en metro (40-50 minutos). Hoy Juan parte de casa entre las 8:10 y las 8:20 y Luis llega al trabajo entre las 9:00 y las 9:10. Además, sabemos que Juan llegó al trabajo entre 10 y 20 minutos después de que Luis saliera de casa.Esta información se puede modelar mediante un CSP métrico entre puntos, representado en la red temporal de la Figura 10.19:

- **Variables:** $T_0, T_1, T_2, T_3, T_4 \in \{0, \ldots, 3600\}$
- **Restricciones:**
  - $10 \leq T_1 - T_0 \leq 20$
  - $30 \leq T_2 - T_1 \leq 40 \vee 60 \leq T_2 - T_1$
  - $10 \leq T_2 - T_3 \leq 20$
  - $20 \leq T_4 - T_3 \leq 30 \vee 40 \leq T_4 - T_3 \leq 50$

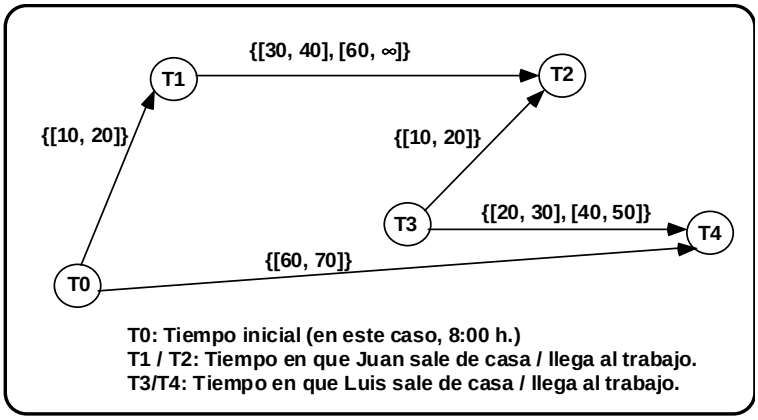


Figura 10.19: Una Red Temporal.

Las restricciones representadas en la Figura 10.19 corresponden a la información explícitamente conocida del problema. Ante ella, se pueden hacer las siguientes preguntas: ¿Esta información es consistente?, es decir, ¿tiene solución?, ¿Es posible que Juan haya usado el tren y Luis haya usado el Metro?, ¿Cuales son los posibles tiempos en los que Luis pudo haber salido de casa?, ¿Quién llega antes al trabajo, Juan o Luis?, etc.

Estas preguntas se pueden responder mediante los procesos inferenciales descritos en la Sección 10.4.2. Particularmente, resulta útil la adaptación del algoritmo de senda consistencia descrito en Algoritmo 10.3 de forma que la clausura de las restricciones en cada subred de 3 nodos (función *Revisar*) se realice mediante la operación de combinación ($\otimes$). Ello da lugar al Algoritmo de Clausura Transitiva (*Transitive Closure Algorithm*, TCA) cuya función *Revisar* se muestra en el Algoritmo 10.4.

Algoritmo 10.4 Función Revisar del Algoritmo de Clausura Transitiva

función $Revisar(i, j, k)$
inicio
$cambiado := false$;
$c'_{ij} \leftarrow c_{ij} \oplus (c_{ik} \otimes c_{kj})$
si $c_{ij} = \emptyset$ **entonces**
  $Inconsistente$;
fin si
si $c'_{ij} \subset c_{ij}$ **entonces**
  $c_{ij} := c'_{ij}$; ; Acota la restricción c_{ij}
  $cambiado := true$
fin si
devolver *cambiado*;
fin *Revisar*;

Como se puede apreciar, el algoritmo TCA obtiene una red senda-consistente, donde:

$$\forall c_{ij}, c_{ik}, c_{kj} \subseteq CSP : c_{ij} \leftarrow c_{ij} \oplus (c_{ik} \otimes c_{kj})$$

La aplicación de este proceso al CSP de la Figura 10.19, obtendría una red clausurada en la que se han hecho explícitas las restricciones implícitamente contenidas en el problema (así fue el caso de la red de la Figura 10.10). La red mínima equivalente se muestra en la Figura 10.20, donde las nuevas restricciones derivadas permitirían responder a algunas de las preguntas anteriores.

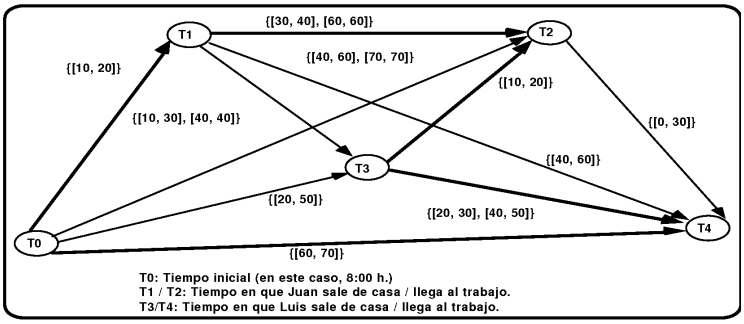


Figura 10.20: Red Mínima correspondiente a la red de la Figura 10.19.

Especialmente en el marco del dominio temporal, y más concretamente en su aplicación a problemas que evolucionan (planificación, scheduling, etc.), aparecen los *CSP Dinámicos* (*DCSP*). Un DCSP se define como una secuencia de CSP, donde cada uno de ellos corresponde a un cambio en el CSP previo [Dechter y Dechter, 1988]. Así, en un DCSP pueden añadirse o retractarse restricciones de forma dinámica, lo cual puede ocurrir durante el proceso de comprobación de la consistencia, durante la búsqueda de soluciones, o bien una vez obtenida una solución. Todos los CSP estáticos en la secuencia de un DCSP deben ser resueltos para solucionar el DCSP. El problema particular en los DCSP reside en encontrar sucesivamente, si existen, nuevas soluciones después de cada modificación. Esto es denominado el *problema de mantenimiento de la solución*, que hace uso de técnicas específicas (obtención de soluciones próximas, reparación de conflictos, etc.) a fin de evitar sucesivos, redundantes e ineficientes, procesos de búsqueda.

10.6.4 Otras extensiones

Para la resolución de un CSP pueden aplicarse también *técnicas estocásticas*, técnicas de búsqueda no sistemáticas e incompletas, incluyendo heurísticas. Ejemplos de este tipo de técnicas son hill-climbing, búsqueda tabú, enfriamiento simulado, algoritmos genéticos, etc. Estas técnicas pueden considerarse como adaptativas, en el sentido de que ellas comienzan su búsqueda en un punto aleatorio del espacio de búsqueda y lo modifican repetidamente, utilizando heurísticas, hasta que se alcanza una solución tras un cierto número de iteraciones. Estos métodos son, generalmente, robustos y

buenos para encontrar soluciones optimizadas en espacios de búsqueda grandes, y han probado su utilidad en la resolución de CSP complejos. La característica estocástica de estos métodos introduce un aspecto aleatorio y pueden ocurrir situaciones donde el proceso de búsqueda se encuentre en porciones erróneas del espacio de búsqueda. Esto generalmente requiere que el sistema reinicie su ejecución empezando en otro punto de partida aleatorio.

Otra extensión de los CSP lo constituyen los *CSP difusos* (*FCSP*) [Dubois y otros, 1993]) que resultan de utilidad en dominios en los que no hay una descripción suficientemente concreta de las restricciones (por ejemplo: "La acción duró *casi* una hora"). Haciendo uso de los conceptos de la lógica difusa, en los FCSP se introduce el concepto de restricción difusa, como una restricción a la que se asocia un función de pertenencia ρ_R. El concepto de satisfabilidad resulta ahora una cuestión de grado. Una asignación a las variables $((x_1, a_1), \ldots, (x_i, a_i))$ es una ρ-solución ($\rho \in [0, 1]$) si satisface la combinación de todas las restricciones con grado ρ. De esta forma, puede relajarse el grado de cumplimiento de restricciones que impiden la obtención de soluciones.

Además del concepto de restricciones difusas, también podría hablarse de restricciones fuertes y restricciones débiles (*hard and soft constraints*). Mientras que las restricciones fuertes deben cumplirse, cabe incluir preferencias o alguna relajación en el cumplimiento de las soft-constraints. Una forma de aproximación a este tipo es el uso de CSP valuados (*Weigthed-CSP*) [Schiex y otros, 1995]. En ellos, las relaciones entre variables incluyen funciones de coste que expresan el grado de satisfacción de las restricciones que vinculan dichas variables.

Otra extensión compleja de los CSP son los denominados *CSP numéricos*. Generalmente n-arios, los CSP numéricos están ligados a una modelización matemática de los problemas (restricciones numéricas, polinomiales, etc.), incluyen dominios infinitos, y requieren la aplicación de técnicas matemáticas específicas para su resolución. Sin embargo, los algoritmos tipo *simplex* son aplicables para restricciones lineales, pero en cambio no resultan tan adecuados para restricciones no lineales o con dominios discretos.

Existe una clara relación de la metodología CSP con la *programación por restricciones* donde las relaciones entre las variables también pueden establecerse en forma de restricciones embebidas en un lenguaje de programación y, particularmente, con la programación lógica. De esta forma, la *programación lógica con restricciones* (Constraint Logic Programming, CLP) puede verse como un CSP donde las restricciones se reescriben como predicados y la unificación en LP se sustituye por la resolución por coerción en un dominio específico (satisfacción de restricciones) [Hentenryck, 1989]. En relación con ello, en el *problema de satisfabilidad* (SAT), las variables se instancian en el dominios de los valores booleanos $\{True, False\}$ y las relaciones utilizan los operadores lógicos *and*, *or* y *not*. En su caso más simple, el problema es encontrar una asignación a las variables que haga cierta una expresión.

10.7 Conclusiones

En este capítulo se ha hecho una revisión de las principales técnicas que subyacen en la metodología CSP, para la resolución de un amplio tipo de problemas combinatorios. Las etapas fundamentales en la resolución de problemas CSP son (Figura 10.21):

1. La modelización del problema, mediante variables, dominios y restricciones, donde es importante la capacidad expresiva de las restricciones a fin de que puedan captar toda la información relevante del problema.

2. La aplicación de métodos inferenciales, que deducen nueva información (nuevas restricciones implícitas). Ello permite poder contestar a preguntas sobre el problema, así como acotar el espacio de búsqueda, por lo que algún proceso de k-consistencia suele realizarse antes del proceso de búsqueda. En las técnicas inferenciales, es importante su capacidad deductiva versus coste computacional. Dependiendo del proceso se obtienen diversos niveles de consistencia, así como mayor información deducida.

3. La aplicación de técnicas de búsqueda de soluciones al problema. Con un coste, en general, *NP-completo* (*NP-duro* en el caso de optimización) estas técnicas se integran con las técnicas inferenciales (técnicas híbridas), y pueden incluir heurísticas, de ordenación de variables o de valores, que permitan la obtención de soluciones en un tiempo razonable.

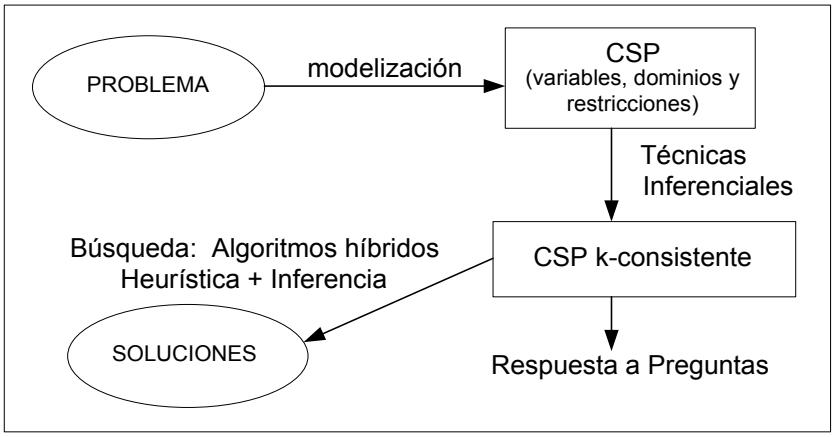


Figura 10.21: Etapas para la resolución de problemas mediante CSP.

Tras la revisión de los principales conceptos y algoritmos de cada una de estas etapas, se ha presentado una breve referencia a algunas de las extensiones o tipologías concretas de CSP. El conocimiento de las técnicas CSP permite disponer de un conjunto de técnicas que permiten abordar la resolución de un amplio tipo de problemas.

La aplicación de los CSP es muy amplia, especialmente en la ingeniería, medicina e informática. Esta metodología resulta útil para solucionar problemas combinatorios, temporales, de configuración etc., cuya resolución esté sujeta a la satisfacción de un conjunto de restricciones entre las variables del problema. Han demostrado su utilidad en aplicaciones de investigación operativa (generación de horarios, scheduling, etc.), bioinformática (identificación y ordenación de secuencias genéticas), ingeniería electrónica (diseño de circuitos), telecomunicaciones (asignación de frecuencias), informática (bases de datos y sistemas de recuperación de la información, lenguaje natural, planificación, etc.), en problemas de diseño y configuración, empaquetamiento, etc. En [Wallace, 1995] pueden verse algunas de las técnicas asociadas a estas aplicaciones.

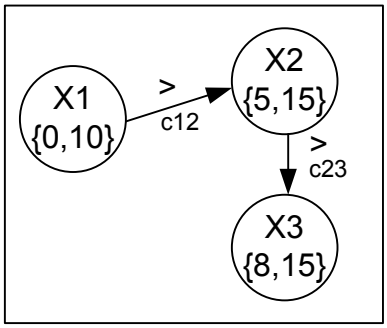
La resolución de problemas mediante un CSP puede considerarse como un método fundamentalmente declarativo. De esta forma, basta especificar el problema a resolver como un CSP. Para resolver un CSP y obtener soluciones pueden utilizarse alguna de las múltiples herramientas disponibles, tanto comerciales, software libre, o librerías con procedimientos especializados en el manejo de restricciones. Estas herramientas existen en diferentes lenguajes, particularmente C++, Java o Lisp. Las herramientas CSP permiten la edición de la especificación del problema (definición de las variables, sus dominios y restricciones) e incluyen potentes métodos de inferencia y búsqueda que nos devuelven una, varias, o todas las soluciones al problema.

10.8 Lecturas Recomendadas

Un texto completo sobre la metodología CSP se encuentra en [Dechter, 2003]. También son interesantes las referencias [Apt, 2003; Frühwirth y Abdennadher, 2003; Marriot y Stuckey, 1998]. De forma más resumida, aunque cubriendo una amplia tipología de CSP y sus aplicaciones, cabe consultar la monografía contenida en [Barber, 2003]. Por otra parte, bastante completa y frecuentemente renovada resulta 'On-Line Guide To Constraint Programming' de R. Barták, accesible en [http://kti.mff.cuni.cz/ bartak/constraints/index.html]. En relación con ello, una visión general de los diferentes tipos de CSP y sus procesos de consistencia puede verse en [Barták, 2001]. Una revisión de las aplicaciones CSP está contenida en [Wallace, 1995]. Finalmente, cabe destacar el interés de diversas páginas web de grupos de trabajo en el área, con aplicaciones, herramientas, benchmarks, bibliografía, tutoriales, etc.

Ejercicios Resueltos

10.1. Dada la red inicial de la figura, donde en cada nodo se representa el dominio de la variable, obténgase la red arco consistente equivalente.



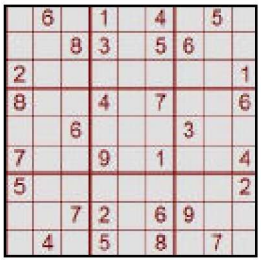
Solución: Para obtener una red arco-consistente, debemos forzar la arco-consistencia de cada arco de la red, $C = \{c_{12}, c_{23}\}$, mediante el Algoritmo descrito en la sección 10.4.2.2, teniendo en cuenta que cada arco implica un arco simétrico.

De esta forma, al hacer c_{12} arco consistente, se limita $d_1 = \{6, 10\}$ y $d_2 = \{5, 9\}$. Se añadiría c_{23} a C, pero ya está.

Para hacer c_{23} arco consistente, se limita $d_2 = \{9\}$ y $d_3 = \{8\}$. Se añade c_{12} a C. Para hacer c_{12} arco consistente, $d_1 = \{10\}$.

Puede verse que los dominios de la red quedan tan acotados que ya no es preciso un proceso de búsqueda para encontrar la solución $x_1 = 10$, $x_2 = 9$, $x_3 = 8$. Por ello, se remarca la utilidad de realizar un preproceso de k-consistencia previo al proceso de búsqueda.

10.2. El sudoku es un claro ejemplo de representación del puzzle como un problema de satisfacción de restricciones. Este problema se publicó en Nueva York en el año 1979 bajo el nombre de "Number Place" y se hizo popular en Japón con el nombre de sudoku ("*Su*dji wa *doku*shin ni kagiru": "los números deben ser sencillos" o "los números deben aparecer una vez"). En el problema del sudoku (ver un ejemplo en la figura), hay que rellenar las casillas de un tablero de 9 × 9 con números del 1 al 9, de forma que no se repita ningún número en la misma fila, columna, o subcuadro de 3 × 3 que componen el sudoku. Modelar este problema como un CSP.



Solución: Este problema se puede modelar como un CSP de forma muy sencilla. Cada celda del tablero se puede ver como una variable, cuyo dominio son los valores naturales entre el 1 y el 9. Por la tanto, existirán 9 x 9 variables. La especificación del CSP es:

- **Variables**: $(x_{11}, x_{12}, \ldots, x_{19}, x_{21}, x_{22}, \ldots, x_{29}, \ldots, x_{91}, x_{92}, \ldots, x_{99})$
- **Dominios**: Para todas las variables, el dominio es: $\{1, 2, \ldots, 9\}$
- **Restricciones**: Debe plantearse una restricción binaria de desigualdad entre:
  - Todos los pares de variables en cada fila:
$\neq (x_{11}, x_{21}, x_{31}, \ldots, x_{91}), \neq (x_{12}, x_{22}, x_{32}, \ldots, x_{92}), \ldots, \neq (x_{19}, x_{29}, x_{39}, \ldots, x_{99})$
  - Todos los pares de variables en cada columna:
$\neq (x_{11}, x_{12}, \ldots, x_{19}), \neq (x_{21}, x_{22}, \ldots, x_{29}), \ldots, \neq (x_{91}, x_{92}, \ldots, x_{99})$
  - Y todos los pares de variables en cada submatriz.
$\neq (x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}), \ldots$ idem para las restantes submatrices.

10.3. Cinco casas consecutivas tienen colores diferentes y son habitadas por hombres de diferentes nacionalidades. Cada uno tiene un animal diferente, una bebida preferida y fuma una marca determinada. Además, se sabe que:

1. *El noruego vive en la primera casa*
2. *La casa de al lado del noruego es azul*
3. *El habitante de la tercera casa bebe leche*
4. *El inglés vive en la casa roja*
5. *El habitante de la casa verde bebe café*
6. *El habitante de la casa amarilla fuma Kools*
7. *La casa blanca se encuentra justo después de la verde*
8. *El español tiene un perro*
9. *El ucraniano bebe té*
10. *El japonés fuma Cravens*
11. *El fumador de Old Golds tiene un caracol*
12. *El fumador de Gitanes bebe vino*
13. *El vecino del fumador de Chesterfields tiene un reno*
14. *El vecino del fumador de Kools tiene un caballo*

Solución: Este problema admite múltiples especificaciones. Podemos definir variables de la siguiente forma, por ejemplo, para la casa 1:

- Casa 1- nacionalidad $\in$ {noruego, ingles, ucraniano, japonés, español}
- Casa 1-animal $\in$ {perro, caracol, reno, caballo, cebra}
- Casa 1- fuma $\in$ {kools, cravens, golds, gitanes, chesterfields}
- Casa 1-bebida $\in$ {leche, café, té, vino, agua}
- Casa 1- color $\in$ {azul, roja, verde, amarilla, blanca}

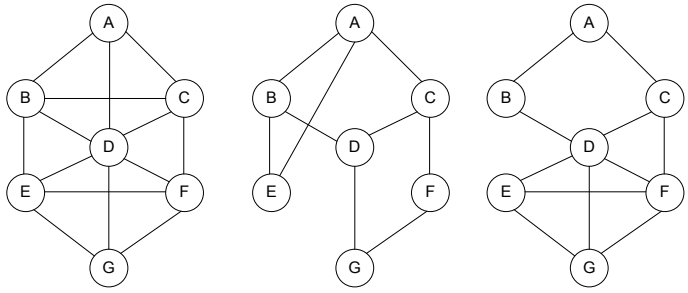
Y sucesivamente para el resto de las casas. Sin embargo, esta definición de variables haría muy compleja la especificación de las restricciones. Otra definición más adecuada sería:

- **Variables:** azul, roja, verde, amarilla, blanca, noruego, ingles, ucraniano, japonés, español, perro, caracol, reno, caballo, cebra, leche, café, té, vino, agua; kools, cravens, golds, gitanes, chesterfields.
- **Dominios:** Todas las variables se instancian en el dominio $\{1, 2, 3, 4, 5\}$, indicando la casa correspondiente.
- **Restricciones:**
  R_1: noruego = 1 ; R_2: azul = noruego + 1 $\vee$ azul + 1 = noruego;
  R_3: leche = 3; R_4: ingles = roja ; R_5: verde = café;
  R_6: amarilla = kools; R_7: blanca = verde + 1; R_8: español = perro;
  R_9: ucraniano = té; R_{10}: japonés = craven; R_{11}: golds = caracol;
  R_{12}: gitanes = vino; R_{13}: chesterfields = reno + 1 $\vee$ chesterfields + 1 = reno;
  R_{14}: kools = caballo +1 $\vee$ kools + 1 = caballo;
  Y, adicionalmente:
  $\neq$ (azul, roja, verde, amarilla, blanca);
  $\neq$ (noruego, ingles, ucraniano, japonés, español);
  $\neq$ (perro, caracol, reno, caballo, cebra);
  $\neq$ (leche, café, té, vino, agua);
  $\neq$ (kools, cravens, golds, gitanes, chesterfields);

Con esta modelización de variables, ha sido fácil especificar las restricciones. Una buena especificación del CSP es importante respecto a la simplicidad de la representación, así como a la eficiencia en la resolución del problema.

Ejercicios Propuestos

10.1. Obtener la ordenación de variables según los diferentes métodos de ordenación estáticos vistos en la Sección 10.5.1 para cada uno de los CSP binarios siguientes:



10.2. En un pueblo viven 4 familias A, B, C y D en casas próximas cuyos números son: 1, 2, 3 y 4. D vive en una casa con menor número que B. B vive próximo

a A en una casa con mayor número. Hay al menos una casa entre B y C. D no vive en una casa cuyo número es 2. C no vive en una casa cuyo número es 4. Modele el problema como un CSP, aplique diversos niveles de consistencia, y resuelva qué familia vive en cada casa aplicando Forward-Checking y Full Look-Ahead.

10.3. Un cuadrado mágico es la disposición de una serie de números enteros en un cuadrado o matriz de tal forma que la suma de los números por columnas, filas y diagonales sea la misma, la constante mágica. Usualmente los números empleados para rellenar las casillas son consecutivos, de 1 a n^2, siendo n el número de columnas y filas del cuadrado mágico (en el ejemplo, $n = 3$). Hacer un CSP que obtenga cuadrados mágicos con diversos valores de n.

$$\text{Numero Magico(n)} = \frac{n(n^2+1)}{2}$$

4	9	2
3	5	7
8	1	6

10.4. Seis amigos y sus respectivas esposas, se hospedan en el mismo hotel; y todos ellos salen todos los días, asistiendo a reuniones de distinto volumen y composición. Para asegurar la variedad en estas salidas, han acordado establecer las siguientes reglas: "Si Antonio está con su mujer, es decir en la misma reunión que su mujer, y David con la suya, y Luis con la señora de Pedro, Enrique debe estar con la señora de Ramón. Si Antonio está con su mujer y Pedro con la suya, y David con la señora de Enrique, Ramón no debe estar con la señora de Luis. Si Enrique y Ramón y sus mujeres están todos en la misma reunión, y Antonio no está con la señora de David, Luis no debe estar con la señora de Pedro. Si Antonio está con su mujer y Ramón con la suya, y David no está con la señora de Enrique, Luis debe estar con la señora de Pedro. Si Luis está con su mujer y Pedro con la suya y Enrique con la señora de Ramón, Antonio no debe estar con la señora de David. Si David y Enrique y sus mujeres están todos en la misma reunión, y Luis no está con la señora de Pedro, Ramón debe estar con la señora de Luis". Modelar dicho problema como un CSP y obtener las posibles combinaciones de reuniones para cada amigo. ¿Es posible que todos los días haya al menos un matrimonio cuyos miembros no estén juntos en la misma reunión? Este problema y el siguiente son originales de Lewis Carroll, en su obra Lógica Simbólica.

10.5. Cinco amigos, Bernardo, Casimir, Luis, Carlos y Marcial, se encuentran cada día en el restaurante. Tienen estas reglas, que observan cada vez que comen: "Bernardo toma sal si y solamente si Casimir toma solo sal o solo mostaza. Bernardo toma mostaza si y solamente si, o bien Luis no toma sal ni mostaza,

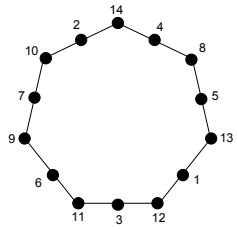
o bien Marcial toma ambas. Casimir toma sal si y solamente si, o Bernardo toma solamente uno de los dos condimentos, o bien Marcial no toma ninguno de ellos. Toma mostaza si y solamente si Luis o Carlos toman dos condimentos. Luis toma sal si y solamente si o bien Bernardo no toma ningún condimento, o bien Casimir toma ambos. Luis toma mostaza si y solamente si Carlos o Marcial no toman ni sal ni mostaza. Carlos toma sal si y solamente si Bernardo o Luis no toman ni sal ni mostaza. Toma mostaza si y solamente si Casimir o Marcial no toman ni sal, ni mostaza. Marcial toma sal si y solamente si Bernardo o Carlos toman los dos condimentos. Marcial toma mostaza si y solamente si Casimir o Luis toman solo un condimento". Modelar el CSP correspondiente. El problema consiste en descubrir si estas reglas son compatibles y, en caso de que lo sean, cuales son las combinaciones posibles.

10.6. Representar mediante un CSP la siguiente información: "Juana, Pepa y Paloma nacieron y viven en ciudades diferentes (Málaga, Madrid y Valencia). Además, ninguna vive en la ciudad donde nació. Juana es más alta que la que vive en Madrid. Paloma es cuñada de la que vive en Valencia. La que vive en Madrid y la que nació en Málaga tienen nombres que comienzan por distinta letra. La que nació en Málaga y la que vive ahora en Valencia tienen nombres que comienzan por la misma letra". ¿Dónde nació y vive cada una?

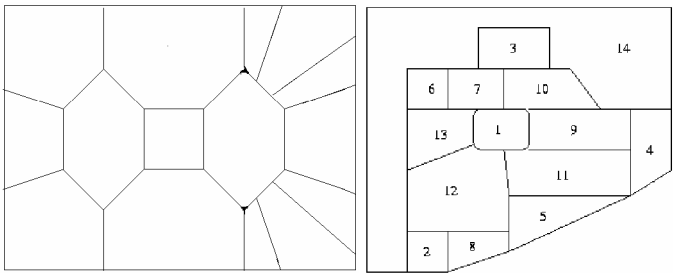
10.7. Obtener el CSP correspondiente al siguiente ejemplo de scheduling: Existen 3 periódicos (P_1, P_2, P_3) y 4 lectores (L_1, L_2, L_3, L_4), que desean leer los periódicos en el mismo orden y con la duración indicada en (P_1, P_2, P_3). Todos deben empezar a partir del ready-time y acabar antes del due-time, según la tabla siguiente:

	Ready-Time	P_1	P_2	P_3	Due-Time
L_1	0	5'	10'	2'	30'
L_1	0	2'	6'	5'	20'
L_3	0	10'	15'	15'	60'
L_4	0	3'	5'	5'	15'

10.8. Henry Dudeney (1847-1930) era un ingenioso inventor de problemas matemáticos. Entre sus aportaciones se encuentra un puzzle con cierta complejidad de resolución. Se trata de un heptágono donde en cada arista hay que colocar tres números: uno en cada vértice y otro en el centro de la arista. Hay que colocar los números del 1 al 14 alrededor de las aristas del heptágono de manera que el número de la arista y los dos vértices extremos sumen lo mismo, para todas las aristas. Modelar el CSP correspondiente.



10.9. Obtener, según las heurísticas de ordenación de variables vistas en la Sección 10.5.1, la mejor ordenación para el coloreado de los siguientes mapas:



Bibliografía

ALLEN, J.: «Maintaning knowledge about temporal intervals». *Comm. Of the ACM*, 1983, **26(1)**, pp. 832–843.

APT, K: *Principles of constraint programming.* Cambridge University Press, 2003. ISBN 0-521-82583-0.

BACCHUS, FAHIEM: «Extending Forward Checking.» En: *CP*, pp. 35–51, 2000.

BARBER, F.: «Monografía: Problemas de Satisfacción de Restricciones». *Revista Iberoamericana de Inteligencia Artificial*, 2003, **20**.

BARTÁK, R.: «Theory and Practice of Constraint Propagation». En: *Proc. of the 3rd Workshop on Constraint Programming for Decision and Control (CPDC2001), June 2001*, pp. 7–14, 2001.

DECHTER, R.: *Constraint Processing.* Morgan Kaufmann Publishers, 2003.

DECHTER, R.; MEIRI, I. y PEARL, J.: «Temporal constraint networks». *Artificial Intelligence*, 1991, pp. 61–95.

DECHTER, RINA y DECHTER, AVI: «Belief maintenance in dynamic constraint networks». En: *Proc.of AAAI-88 (St. Paul, MN)*, pp. 37–42, 1988.

DENT, MICHAEL J. y MERCER, ROBERT E.: «Minimal Forward Checking.» En: *ICTAI*, pp. 432–438, 1994.

DUBOIS, D.; FARGIER, H. y PRADE, H.: «The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction». En: *IEEE Int. Conf. on Fuzzy Systems*, pp. 1131–1136, 1993.

FALTINGS, B. y YOKOO, M.: «Distributed Constraint Satisfaction». *Special Issue on Artificial Intelligence*, 2005, **161(1-2)**.

FREUDER, E. C.: «A sufficient condition for backtrack-free search». *Journal of the ACM*, 1982, **29(1)**, pp. 24–32.

FRÜHWIRTH, THOM y ABDENNADHER, SLIM: *Essentials of constraint programming.* Springer, 2003. ISBN 3-540-67623-6.

FROST, DANIEL y DECHTER, RINA: «Dead-End Driven Learning». En: *Proceedings of the Twelfth National Conference of Artificial Intelligence(AAAI-94)*, volumen 1, pp. 294–300. AAAI Press, Seattle, Washington, USA. ISBN 0-262-61102-3, 1994. `citeseer.ist.psu.edu/frost94deadend.html`

GASCHNIG, J.: «Performance measurement and analysis of certain search algorithms». *Informe técnico CMU-CS-79-124*, Carnegie-Mellon University, 1979.

HENTENRYCK, P. V.: *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

MACKWORTH, A. K.: «Consistency in networks of relations». *Artificial Intelligence*, 1977, **8**, pp. 121–118.

MARRIOT, KIM y STUCKEY, PETER J.: *Programming with Constraints*. MIT Press, 1998. ISBN 0-262-13341-5.

MONTANARI, U.: «Networks of constraints: fundamental properties and applications to picture processing». *Information Science*, 1974, **7**, pp. 95–132.

PROSSER, P.: «Hybrid algorithms for the constraint satisfaction problem». *Computational Intelligence*, 1993, **9(3)**, pp. 268–299.

SALIDO, M. A. y BARBER, F.: «Distributed CSPs by Graph Partitioning». *Applied Mathematics and Computation*, 2006, **183(1)**, pp. 491–498.

SCHIEX, T.; FARGIER, H. y VERFAILLE, G.: «Valued constraint satisfaction Problems». En: *IJCAI-95*, pp. 631–637, 1995.

STERGIOU, K. y WALSH, T.:. «Encoding of non-binary constraints satisfaction problems».

VAN BEEK, P.: «Temporal query processing with indefinite information». *Artificial Intelligence in Medicine*, 1991, **3(6)**, pp. 325–339.

VILAIN, M. y KAUTZ, H.: «Constraint propagation algorithms for temporal reasoning». En: *Procc. of the AAAI-86*, pp. 377–382, 1986.

WALLACE, G.: «Survey: Practical Applications of Constraint Programming». *Constraints Journal*, 1995, **1(1)**, pp. 139–168.

YOKOO, M.; DURFEE, E.H.; ISHIDA, T. y KUWABARA, K.: «The distributed constraint satisfaction problem: formalization and algorithms». *IEEE Transactions on Knowledge and Data Engineering*, 1998, **10(5)**, pp. 673–685.