Introducción a la Programación de Restricciones

Federico Barber*, **Miguel A. Salido**†

*Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, Spain
fbarber@dsic.upv.es

†Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Alicante, Spain
msalido@dccia.ua.es

Resumen

La programación de restricciones es una tecnología software utilizada para la descripción y posterior resolución efectiva de grandes y complejos problemas, particularmente combinatorios, de muchas áreas de la vida real. Muchos de estos problemas pueden modelarse como problemas de satisfacción de restricciones (CSPs) y resolverse usando técnicas de programación de restricciones. Esto incluye problemas de áreas tales como inteligencia artificial, investigación operativa, bases de datos, sistemas expertos, etc. Algunos ejemplos son scheduling, planificación, razonamiento temporal, diseño en la ingeniería, problemas de empaquetamiento, criptografía, diagnosis, toma de decisiones, etc. El manejo de este tipo de problemas es NP [26]. En este artículo introductorio se presenta una introducción de los conceptos, algoritmos y técnicas más relevantes en el área de CSPs que servirá para que el lector tenga un conocimiento global de los CSPs así como una notación general que servirá para comprender mejor los siguientes trabajos presentados en esta monografía.

Palabras clave: Problemas de Satisfacción de Restricciones, Inteligencia Artificial, Optimización.

1. Introducción

Hoy en día, muchas decisiones que tomamos a la hora de resolver nuestros problemas cotidianos están sujetos a restricciones. Problemas tan cotidianos como fijar una cita con unos amigos, comprar un coche o preparar una paella puede depender de muchos aspectos interdependientes e incluso conflictivos, cada uno de los cuales está sujeto a un conjunto de restricciones. Además, cuando se encuentra una solución que satisface plenamente a unos, puede que no sea tan apropiada para otros, por lo que a veces no es suficiente con obtener una única solución.

Durante los últimos años, la programación de restricciones ha generado una gran expectación entre expertos de muchas áreas debido a su potencial para la resolución de grandes problemas reales. Por ello, no es de sorprender, que la ACM (Association for Computer Machinery) ha identificado a la programación de restricciones como una de las direcciones estratégicas en la investigación informática [4]. Sin embargo, al mismo tiempo, se considera la programación de restricciones 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 área del pro-

blema y consecuentemente encontrar soluciones 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]

El Origen de la Programación de Restricciones

Los primeros trabajos relacionados con la programación de restricciones datan de los años 60 y 70 en el campo de la Inteligencia Artificial.

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 con dominios finitos, mientras que la resolución de restricciones está orientada principalmente a problemas sobre dominios infinitos o dominios más complejos. En este artículo nos centraremos principalmente en los problemas de satisfacción de restricciones, que básicamente consiste en un conjunto finito de variables, un dominio de valores para cada variable y un conjunto de restricciones que acotan la combinación de valores que las variables pueden tomar. Así, el objetivo es encontrar un valor para cada variable de manera que se satisfagan todas las restricciones del problema. En general, la obtención de soluciones en este tipo de problemas NP-completo. Adicionalmente, la obtención de soluciones optimizadas es en general NP-dura.

Ejemplo. El problema de coloración del mapa es un problema clásico que se puede formular como un CSP. En este problema, hay un conjunto de colores y queremos colorear cada región del mapa de manera que las regiones adyacentes tengan distintos colores. En la formulación del CSP, hay 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. Dicho mapa puede ser representado mediante un grafo donde los nodos son variables que representan a los colores asociados a las regiones y cada par de regiones adyacentes están unidas por una arista. En la Figura 1 se muestra el ejemplo del problema de coloración del mapa. Seleccionamos cuatro regiones x,y,z,w para ser coloreadas. Cada región del mapa se corresponde con una variable en el grafo. Si asumimos que cada región puede colorearse con uno de los tres colores, rojo (r), verde (v) y azul (a), entonces cada variable del grafo tiene tres posibles valores {r,v,a}.

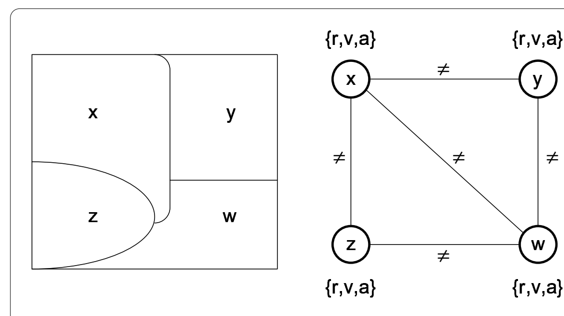

Figura 1. Problema de coloración del mapa

Las restricciones de este problema expresan que regiones adyacentes tienen que ser coloreadas con diferentes colores. En la representación en el grafo, las variables correspondientes a regiones adyacentes están conectadas por una arista. Hay cinco restricciones en el problema, es decir, cinco aristas en el grafo. 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.

2. Resolución del CSP

La resolución de un problema de satisfacción de restricciones (CSP) consta de dos fases diferentes:

- modelar el problema como un problema de satisfacción de restricciones. La modelización expresa el problema mediante una sintaxis de CSPs, es decir, mediante un conjunto de variables, dominios y restricciones del CSP. En la sección 3 presentamos la modelización de un CSP.
- procesar el problema de satisfacción de restricciones resultante. Una vez formulado el problema como un CSP, hay dos maneras de procesar las restricciones:
  1. *Técnicas de consistencia.* Se trata de técnicas para la resolución de CSPs basadas en la eliminación de valores inconsistentes de los dominios de las variables.

2. *Algoritmos de búsqueda*. Estos algoritmos se basan en la exploración sistemática del espacio de soluciones hasta encontrar una solución o probar que no existe tal solución.

Las técnicas de consistencia o inferenciales permiten deducir información del problema, (niveles de consistencia, valores posibles de variables, dominios mínimos, etc.), aunque en general se combinan con las técnicas de búsqueda, ya que reducen el espacio de soluciones y los algoritmos de búsqueda exploran dicho espacio resultante.

3. Modelización del CSP

Generalmente la declaración de un problema se suele expresar de muchas maneras diferentes, e incluso en lenguaje natural. Una parte muy importante para la resolución de problemas de la vida real es el modelado del problema en términos de CSPs, es decir, variables, dominios y restricciones.

A continuación consideraremos un ejemplo en el cual veremos distintas modelizaciones de un mismo problema y las ventajas de una modelización adecuada para resolverlo.

Consideremos el conocido problema criptográfico 'send+more=money' utilizado en [37]. El problema puede ser declarado como: *asignar a cada letra {s, e, n, d, m, o, r, y} un dígito diferente del conjunto {0,...,9} de forma que se satisfaga send+more=money.*

La manera más fácil de modelar este problema es asignando una variable a cada una de las letras, todas ellas con un dominio *{0,...,9}* y con las restricciones de que todas las variables toman valores distintos y con la correspondiente restricción para que se satisfaga 'send+more=money'. De esta forma las restricciones (no binarias) 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 (s,e,n,d,m,o,r,y);

Para el algoritmo más general como es Backtracking[1] (BT), este modelo no es muy eficiente porque con BT, todas las variables necesitan ser instanciadas antes de comprobar estas dos restricciones. De esta manera no se puede podar el espacio de búsqueda para agilizar la búsqueda de soluciones. 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.

Veamos a continuación un modelo más eficiente para resolver el problema. Este modelo utiliza los bit de acarreo para descomponer la ecuación anterior en una colección de pequeñas restricciones. Tal y como está planteado el problema, M debe de tomar el valor 1 y por lo tanto S solamente puede tomar valores de $\{1,...,9\}$. Además de las variables del modelo anterior, el nuevo modelo incluye tres variables adicionales, c_1, c_2, c_3 que llamaremos '*portadoras*'. Los dominios de las variables e, n, d, o, r e y son $\{0,...,9\}$, el dominio de s es $\{1,...,9\}$, el dominio de m es $\{1\}$, y los dominios de las variables portadores c_1, c_2, c_3 son $\{0,1\}$. Con la ayuda de las variables portadores, la restricción de la ecuación anterior puede descomponerse en varias restricciones más pequeñas:

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

La ventaja de este modelo es que estas restricciones más pequeñas pueden comprobarse antes en la búsqueda de backtracking, y así podarse muchas inconsistencias.

En las dos formulaciones anteriores, la restricción de todas diferentes puede reemplazarse por un conjunto de pequeñas restricciones, es decir $e \neq s, ..., r \neq y$, obteniendo así un modelo alternativo.

Como hemos visto en las formulaciones anteriores, dependiendo de la transformación que se haga del problema planteado en lenguaje natural a la modelización en forma de CSP, el problema se resolverá con más o menos eficiencia.

[1]Este algoritmo lo detallamos en la siguiente sección

Concretamente, en el contexto de los problemas de satisfacción de restricciones no binarias, cuando tratamos de resolver un CSP no binario ya modelado, nos volvemos a encontrar con un nuevo problema crucial de modelización. ¿Debemos convertir el problema no binario en uno binario, o debemos dejarlo en su formulación original?

4. Conceptos CSP

En esta sección presentamos los conceptos y objetivos básicos que son necesarios en los problemas de satisfacción de restricciones y que utilizaremos a lo largo de esta monografía.

CSP. Un problema de satisfacción de restricciones (CSP) es una terna (X, D, C) donde:

- X es un conjunto de n variables $\{x_1, ..., x_n\}$.
- $D =< D_1, ..., D_n >$ es un vector de dominios. La $i-$ésima componente D_i es el dominio que contiene todos los posibles valores que se le pueden asignar a la variable x_i.
- C es un conjunto finito de restricciones. Cada restricción $n-$aria (c_n) está definida sobre un conjunto de variables $\{x_1, ..., x_n\}$ restringiendo los valores que las variables pueden simultáneamente tomar.

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

Solución. Una solución a un CSP es una asignación de valores a todas las variables de forma que se satisfagan todas las restricciones. 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. Por lo tanto diremos que un problema es consistente, si existe al menos una solución, es decir una tupla consistente $(a_1, a_2, ..., a_n)$.

Básicamente los objetivos que deseamos obtener de un CSP se centran en encontrar:

- una solución, sin preferencia alguna
- todas las soluciones
- una óptima, o al menos una buena solución, dando alguna función objetivo definida en términos de algunas o todas las variables.

4.1. Notación CSP

Antes de entrar con más detalles en los problemas de satisfacción de restricciones, vamos a resumir la notación que utilizaremos a lo largo de esta monografía.

General: El número de variables de un CSP lo denotaremos por n. La longitud del dominio una variable x_i lo denotamos por $d_i = |D_i|$. El número de restricciones totales lo denotaremos por c. La *aridad* máxima de una restricción la denotaremos por k. En el caso de problemas disyuntivos, al número máximo de disyunciones que tiene una restricción disyuntiva lo denotaremos por l.

Variables: Para representar las variables utilizaremos las últimas letras del alfabeto en cursiva, por ejemplo x, y, z, así como esas mismas letras con un subíndice, por ejemplo x_1, x_i, x_j. Estos subíndices son letras seleccionada por mitad del alfabeto o números enteros. Al conjunto de variables $x_i, ..., x_j$ lo denotaremos por $X_{i,...,j}$.

Dominios/Valores: : El dominio de una variable x_i lo denotamos por D_i. A los valores individuales de un dominio los representaremos mediante las primeras letras del alfabeto, por ejemplo, a, b, c, y al igual que en las variables también pueden ir seguidas de subíndices. La asignación de un valor a a una variable x la denotaremos mediante el par (x, a). Como ya mencionamos en la definición (4) una tupla de asignación de variables $((x_1, a_1), ..., (x_i, a_i))$ la denotaremos por $(a_1, ..., a_i)$.

Restricciones: : Una restricción $k-$aria entre las variables $\{x_1, ..., x_k\}$ la denotaremos por $C_{1..k}$. De esta manera, una restricción binaria entre las variables x_i y x_j la denotaremos por C_{ij}. Cuando los índices de las variables en una restricción no

son relevantes, lo denotaremos simplemente por C. El conjunto de variables involucrados en una restricción $C_{i..k}$ lo representaremos por $X_{C_{i..k}}$. El resto de la notación que surja a lo largo de la monografía se definirá cuando sea necesaria.

4.2. Restricciones

En esta sección veremos algunas definiciones sobre restricciones y explicaremos algunas de las propiedades básicas.

La ***aridad*** de una restricción es el número de variables que componen dicha restricción. Una restricción unaria es una restricción que consta de una sola variable. Una restricción binaria es una restricción que consta de dos variables. Una restricción ternaria consta de tres variables. Una restricción no binaria (o $n - aria$) es una restricción que involucra a un número arbitrario de variables.

Ejemplo. 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 $2x_1 - x_2 + 4x_3 \leq 4$ es una restricción ternaria. Por último un ejemplo de restricciones $n-$aria sería $x_1 + 2x_2 - x_3 + 5x_4 \leq 9$.

Una ***tupla*** p de una restricción $C_{i..k}$ es un elemento del producto cartesiano $D_i \times .. \times D_k$. Una tupla p que satisface la restricción $C_{i..k}$ se le llama tupla permitida o válida. Una tupla p que no satisface la restricción $C_{i..k}$ se le llama tupla no permitida o no válida. Una tupla p de una restricción $C_{i..k}$ se dice que es *soporte* para un valor $a \in D_j$ si la variable $x_j \in X_{C_{i..k}}$, p es una tupla permitida y contiene a a en la correspondiente posición de x_j en la restricción.

De esta manera, verificar si una tupla dada es permitida o no por una restricción se llama comprobación de la consistencia. Una restricción puede definirse *extensionalmente* mediante un conjunto de tuplas válidas o no válidas y también *intencionalmente* mediante una función aritmética. En el caso de CSPs continuos es imposible representar las restricciones extensionalmente ya que hay un número infinito de tuplas válidas y no válidas.

Ejemplo. Consideremos una restricción entre 4 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 *intencionalmente* mediante la expresión $x_1 + x_2 \leq x_3 + x_4$. Además, esta restricción también puede representarse *extensionalmente* mediante el conjunto de tuplas permitidas $\{(1,1,1,1), (1,1,1,2), (1,1,2,1), (1,1,2,2), (2,1,2,2), (1,2,2,2), (1,2,1,2), (1,2,2,1), (2,1,1,2), (2,1,2,1), (2,2,2,2)\}$, o mediante el conjunto de tuplas no permitidas $\{(1,2,1,1), (2,1,1,1), (2,2,1,1), (2,2,1,2), (2,2,2,1)\}$.

5. Consistencia en un CSP

Los algoritmos de búsqueda sistemática para la resolución de CSPs 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 CSPs. Una de las principales dificultades con las que nos encontramos en los algoritmos de búsqueda es la aparición de inconsistencias locales que van apareciendo continuamente [26]. Las inconsistencias locales son valores individuales o combinación de valores de las variables que no pueden participar en la solución porque no satisfacen alguna propiedad de consistencia. Por ejemplo, si el valor a de la variable x es incompatible con todos los valores de una variable y que está ligada a x mediante una restricción, entonces a es inconsistente con respecto a la propiedad local de consistencia de arco o arco-consistencia[2]. Por lo tanto si forzamos alguna propiedad de consistencia A podemos borrar todos los valores que son inconsistentes con respecto a la propiedad A. Esto no significa que todos los valores que no pueden participar en una solución sean borrados. Puede haber valores que son consistentes con respecto a A pero son inconsistentes con respecto a cualquier otra consistencia local B. Sin embargo, *consistencia global* significa que todos los valores 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

[2] Arco-Consistencia lo definiremos más adelante.

local causada por ellas y malgastará esfuerzo de búsqueda tratando repetidamente de intentar instanciaciones que ya han sido probadas. Veamos el siguiente ejemplo.

Ejemplo. 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: $y < z$ y $x \neq y$. 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 y y 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.

En la literatura se han propuesto varias técnicas de *consistencia local* como formas de mejorar la eficiencia de los algoritmos de búsqueda. Tales técnicas borran valores inconsistentes de las variables o inducen restricciones implícitas que nos ayudan a podar el espacio de búsqueda. Estas técnicas de consistencia local se usan como etapas de preproceso donde se detectan y se eliminan las inconsistencias locales antes de empezar o durante la búsqueda con el fin de reducir el árbol de búsqueda.

Freuder presentó una noción genérica de consistencia llamada (i, j)-consistencia [16]. Un problema es (i, j)-consistente si cualquier solución a un subproblema con i variables puede ser extendido a una solución incluyendo j variables adicionales. La mayoría de las formas de consistencia se pueden ver como especificaciones de la (i, j)-consistencia. Cuando i es $k - 1$ y j es 1, podremos obtener la k-consistencia [14]. Un problema es *fuertemente k-consistente* si es i-consistente para todo $i \leq k$. Un problema fuertemente k-consistente con k variables se llama *globalmente consistente*. La complejidad espacial y temporal en el peor caso de forzar la k-consistencia es exponencial con k. Además, cuando $k \geq 2$, forzar la k-consistencia cambia la estructura del grafo de restricciones añadiendo nuevas restricciones no unarias. Esto hace que la k-consistencia sea impracticable cuando k es grande. A continuación damos unas definiciones formales de algunas de las consistencias locales que más tarde discutiremos con detalle, basadas en Debruyne y Bessiére [8].

- Un CSP $n-$ario es (i, j)*-consistente* si y solamente si $\forall x_i \in X, D_i \neq \phi$ y cualquier instanciación consistente de i variables puede ser extendido a una instanciación consistente que involucra a j variables adicionales.
- Un CSP binario es *arco-consistente* si y solamente si $\forall a \in D_i, \forall x_j \in X$, con $C_{ij} \in C, \exists b \in D_j$ tal que b es un soporte para a en C_{ij}.
- Un CSP binario es *path-consistente* si y solamente si $\forall x_i, x_j \in X, (x_i, x_j)$ es path-consistente. Un par de variables (x_i, x_j) es *path-consistente* si y sólo si $\forall(a, b) \in C_{ij}, \forall x_k \in X, \exists c \in D_k$ por lo que c es un soporte para a en c_{ik} y c es un soporte para b en c_{jk}.
- Un CSP binario es *fuertemente path-consistente* si y solamente si es $(j, 1)-$consistente para $j \leq 2$.
- Un CSP binario es *inversamente path-consistente* si y solamente si $\forall(x_i, a) \in D, \forall x_j, x_k \in X$ tal que $j \neq i \neq k \neq j$, $\exists(x_j, b) \in D$ y $(x_k, c) \in D$ tal que b es un soporte para a en C_{ij}, c es un soporte para a en C_{ik} y c es un soporte para b en C_{jk}.
- Un CSP binario es *vecino inversa-consistente* si y solamente si $\forall(x_i, a) \in D, (x_i, a)$ se puede extender a una instanciación consistente de los vecinos de x_i.
- Un CSP binario es *path-consistente restringido* si y solamente si $\forall x_i \in X, D_i \neq \phi$, D_i es arco-consistente y, $\forall(x_i, a) \in D_i, \forall x_j \in X$ tal que (x_i, a) tiene un único soporte b en $D_j, \forall x_k \in X$ tal que $\{C_{ik}, C_{jk}\} \in C, \exists c \in D_k$ tal que c es un soporte para a en C_{ik} y c es un soporte para b en C_{jk}.
- Un CSP binario es *simple arco-consistente* si y solamente si $\forall x_i \in X, D_i \neq \phi$ y $\forall(x_i, a) \in D, P \mid_{D_{x_i}=\{a\}}$ tiene un sub-dominio arco-consistente. Denotamos $P \mid_{D_{x_i}=\{a\}}$ como el CSP obtenido restringiendo D_i al valor a en un CSP P, donde $x_i \in X$.
- Un CSP $n-$ario es *arco-consistente generalizado* si y solamente si $\forall D_i \in D, D_i \neq \phi$ y D_i es arco-consistente generalizado. Un dominio D_i es arco-consistente generalizado si y solo si $\forall a \in D_i, \forall x_j, ..., x_k \in X$, con $C_{j,...i,...,k} \in C$, existe una tupla $t = \{b, ..., a, ...c\}$ permitida por $C_{j,...i,...k}$ tal que t es un soporte para (x_i, a) en $C_{j,...i,...k}$.

5.1. Niveles de Consistencia Local

Pueden definirse caso particulares de k-consistencia. Detalles de algoritmos correspondientes pueden verse en [25].

Consistencia de Nodo (1-consistencia)

La consistencia local más simple de todas es la *consistencia de nodo* o *nodo-consistencia.* Forzar este nivel de consistencia nos asegura que todos los valores en el dominio de una variable satisfacen todas las restricciones unarias sobre esa variable.

Así, un problema es *nodo-consistente* si y sólo si todas sus variables son nodo-consistentes:

$$\forall x_i \in X, \forall C_i, \exists a \in D_i : a \text{ satisface } C_i$$

Ejemplo. Consideremos 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. En la Figura 2 mostramos el resultado de aplicar nodo-consistencia a la variable x.

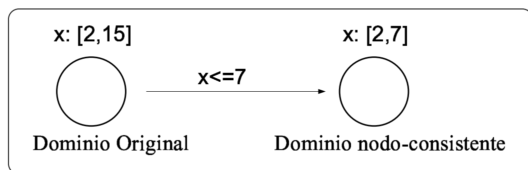

Figura 2. Consistencia de nodo, (nodo-consistencia)

Consistencia de Arco (2-consistencia)

Un problema binario es arco-consistente si para cualquier par de variables restringidas x_i y x_j, para cada valor a en D_i hay al menos un valor b en D_j tal que las asignaciones (x_i, a) y (x_j, b) satisfacen la restricción entre x_i y x_j. Cualquier valor en el dominio D_i de la variable x_i que no es arco-consistente puede ser eliminado de D_i ya que no puede formar parte de ninguna solución. El dominio de una variable es arco-consistente si todos sus valores son arco-consistentes.

Ejemplo. Dada la restricción $C_{ij} = x_i < x_j$ de la Figura 3, podemos observar que el arco C_{ij} es 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 C_{ij}. Sin embargo si la restricción fuese $C_{ij} = x_i > x_j$ no sería arco-consistente.

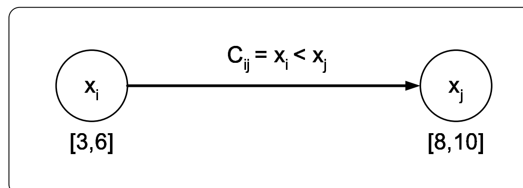

Figura 3. Consistencia de arco, (arco-consistencia)

Así, un problema es *arco-consistente* si y sólo si todos sus arcos son arco-consistentes:

$$\forall\ C_{ij} \in C, \forall a \in D_i, \exists b \in D_j \text{ tal que } b \text{ es un soporte para } a \text{ en } C_{ij}.$$

Consistencia de caminos (3-consistencia)

La consistencia de caminos [29] (Path-consistency) es un nivel más alto de consistencia local que la arco-consistencia. La consistencia de caminos requiere para cada par de valores a y b de dos variables x_i y x_j, que la asignación de a a x_i y de b a x_j satisfaga la restricción entre x_i y x_j, y que además exista un valor para cada variable a lo largo del camino entre x_i y x_j de forma que todas las restricciones a lo largo del camino se satisfagan. Montanari demostró que un CSP satisface la consistencia de caminos si y sólo si todos los caminos de longitud dos cumplen la consistencia de caminos [29]. En esta propiedad se basa el algoritmo *Transitive Closure Algorithm* (TCA) [29][26], el cual obtiene la consistencia de caminos para CSPs binarios, deduciendo nuevas restricciones derivadas. Más concretamente elimina valores de los dominios de las variables, y acotan las restricciones de forma que el CSP satisface la consistencia de caminos.

Así, un problema satisface la consistencia de caminos si y sólo si todo par de variables (x_i, x_j) es *path-consistente*:

$$\forall (a, b) \in C_{ij}, \forall x_k \in X, \exists c \in D_k \text{ tal que } c \text{ es un soporte para } a \text{ en } c_{ik} \text{ y para } b \text{ en } c_{jk}.$$

Cuando un problema satisface la consistencia de caminos y además es nodo-consistente y arco-consistente se dice que satisface fuertemente la consistencia de caminos (strongly path-consistent).

Ejemplo. Dado el problema representado mediante el grafo de restricciones de la Figura 4, podemos observar que el problema satisface la consistencia de caminos ya que cualquier camino entre un par de nodos lo es.

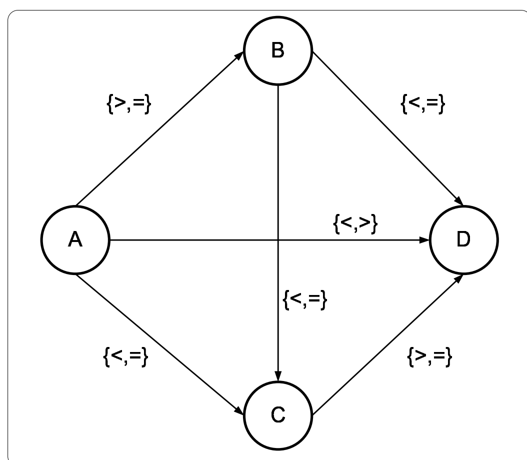

Figura 4. Consistencia de caminos

5.2. Consistencia Global

A veces es deseable una noción más fuerte que la consistencia local. Decimos que un etiquetado, construido mediante un algoritmo de consistencia, es globalmente consistente si contiene solamente aquellas combinaciones de valores que forman parte de al menos una solución.

Dado un CSP (X, D, C), se dice que es ***globalmente consistente*** si y sólo si $\forall x_i \in X, \forall a \in D_i$, $x_i = a$ forma parte de una solución del CSP.

Un etiquetado globalmente consistente es una representación compacta y conservadora de todas las soluciones admitidas por un CSP. Es correcto en el sentido de que el etiquetado 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. En una red de restricciones globalmente consistente la búsqueda puede llevarse a cabo sin backtracking [15].

En general, un etiquetado globalmente consistente puede requerir de forma explícita representar restricciones para todas las variables del problema, es decir, generar etiquetas $n-1-$dimensionales para un problema de talla n forzando la $n-$consistencia. Esta tarea tiene una complejidad exponencial en el peor caso. Para clases especiales de problemas, niveles bajos de consistencia son equivalentes a la consistencia global. Estos resultados permiten a los algoritmos polinómicos llevar a cabo etiquetados globalmente consistentes. Los podemos resumir en los siguientes:

- La Arco-consistencia es equivalente a la consistencia global cuando la red de restricciones es un árbol [15].

- La consistencia de caminos es equivalente a la consistencia global cuando el CSP es convexo[3] y binario [10][36].

6. Algoritmos de Búsqueda

Los algoritmos de Backtracking es la base fundamental de los algoritmos de búsqueda sistemática para la resolución de CSPs. Estos algoritmos buscan a través del espacio de las posibles asignaciones de valores a las variables garantizando encontrar una solución, si es que existe, o demostrando que el problema no tiene solución, en caso contrario. Es por ello por lo que se conocen como algoritmos *completos*. Los algoritmos incompletos, que no garantizan encontrar una solución, también son muy utilizados en problemas de satisfacción de restricciones y en problemas de optimización debido a su mayor eficiencia y el alto coste que requiere una búsqueda completa. Estos algoritmos incluyen algoritmos genéticos, búsqueda tabú, etc.

En la literatura se han desarrollado muchos algoritmos de búsqueda completa para CSPs binarios. Algunos ejemplos son: 'backtracking cronológico', 'backjumping' [20], 'conflict-directed backtracking' [31], 'backtracking dinámico' [22], 'forward-checking' [23], 'minimal forward checking' [11], algoritmos híbridos como forward checking con conflict-directed backtracking [31] y 'manteniendo arco-consistencia' (MAC) [20] [33]. Una revisión de estos algoritmos puede verse en [27].

[3] Un CSP es convexo cuando sus restricciones determinan un espacio de soluciones convexo.

Algunos de los algoritmos anteriores se han extendido a CSP no binarios. Por ejemplo hay varias extensiones de forward checking para problemas no binarios [5]. Además MAC se ha extendido a un algoritmo que mantiene la arco-consistencia generalizada sobre restricciones de cualquier aridad. En esta sección discutimos algunos de los algoritmos de búsqueda más comunes.

7. El Árbol de Búsqueda

Las posibles combinaciones de la asignación de valores a las variables en un CSP genera un espacio de búsqueda que puede ser visto como un árbol de búsqueda. La búsqueda mediante backtracking en un CSP corresponde a la tradicional *exploración primero en profundidad* en el árbol de búsqueda (Figura 5). Si asumimos que el orden de las variables es estático y no cambia durante la búsqueda entonces el nodo en el nivel k del árbol de búsqueda representa un estado donde las variables $x_1, ..., x_k$ están asignadas y el resto $x_{k+1}, ..., x_n$ no lo están. Nosotros podemos asignar cada nodo en el árbol de búsqueda con la tupla consistente de todas las asignaciones llevadas a cabo. La raíz del árbol de búsqueda representa la tupla vacía, donde ninguna variable tiene asignado valor alguno.

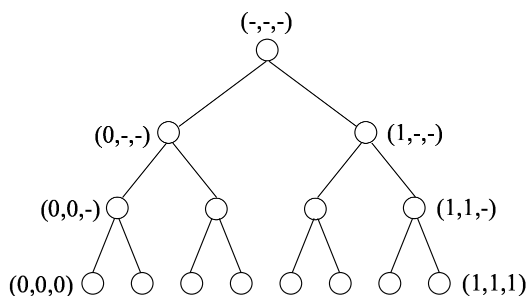

Figura 5. El Árbol de búsqueda

Los nodos en el primer nivel son $1 - tuplas$ que representan estados donde se les ha asignado un valor a la variable x_1. Los nodos en el segundo nivel son $2 - tuplas$ que representan estados donde se le asignan valores a las variables x_1 y x_2, y así sucesivamente. Si n es el número de variables del problema, los nodos en el nivel n, que representan las hojas del árbol de búsqueda, son $n - tuplas$, que representan la asignación de valores para todas las variables del problema. De esta manera, si una $n - tupla$ es consistente, entonces es solución del problema. Un nodo del árbol de búsqueda es consistente si la asignación parcial actual es consistente, o en otras palabras, si la tupla correspondiente a ese nodo es consistente. Los nodos que se encuentran próximos a la raíz, se les llama nodos superficiales. Los nodos próximos a las hojas del árbol de búsqueda se le llaman nodos profundos.

La Figura 5 muestra un ejemplo con tres variables y cuyos dominios son 0 y 1. Cada variable corresponde a un nivel en el árbol. En el nivel 0 no se ha hecho ninguna asignación, por lo que se representa mediante la tupla $(-, -, -)$. Los demás nodos corresponden a tuplas donde al menos se ha asignado una variable. Por ejemplo el nodo hoja que se encuentra a la derecha en el árbol de búsqueda es el nodo $(1, 1, 1)$, donde todas las variables toman el valor 1. En la búsqueda primero en profundidad del árbol de búsqueda, la variable correspondiente al nivel actual se llama variable actual. Las variables correspondientes a niveles menos profundos se llaman variables pasadas. Las variables restantes que se instanciarán más tarde se llaman variables futuras. Más detalles y especificación de algunos de estos algoritmos pueden también verse en esta misma monografía [25].

7.1. Backtracking Cronológico

El algoritmo de búsqueda sistemática para resolver CSPs es el *Algoritmo de Backtracking Cronológico* (BT). Si asumimos un orden estático de las variables y de los valores en las variables, este algoritmo trabaja de la siguiente manera. El algoritmo selecciona la siguiente variable de acuerdo al orden de las variables y le asigna su próximo valor. Esta asignación de la variable se comprueba en todas las restricciones en las que forma parte la variable actual y las anteriores. Si todas las restricciones se han satisfecho, el backtracking cronológico selecciona la siguiente variable y trata de encontrar un valor para ella de la misma manera. Si alguna restricción no se satisface entonces la asignación actual se deshace y se prueba con el próximo valor de la variable actual. Si no se encuentra ningún valor consistente entonces tenemos una situación sin salida (*dead-end*) y el algoritmo retrocede a la variable anteriormente asignada y prueba asignándole un nuevo valor. Si asumimos que estamos buscando una sola solución, el backtracking cronológico finaliza cuando a todas las variables se les ha asignado un valor, en cuyo caso devuelve una solución, o cuando

todas las combinaciones de variable-valor se han probado sin éxito, en cuyo caso no existe solución.

Es fácil ver como el backtracking cronológico puede ser generalizado a restricciones no binarias. Cuando se prueba un valor de la variable actual, el algoritmo comprobará todas las restricciones en las que sólo forman parte la variable actual y las anteriores. Si una restricción involucra a la variable actual y al menos una variable futura, entonces esta restricción no se comprobará hasta que se hayan comprobado todas las variables futuras de la restricción. Por ejemplo, si nosotros tenemos las restricciones $x_1 + x_2 + x_3 \leq 5$ y $x_1 + x_3 + x_4 \leq 0$ y x_3 es la variable actual entonces el backtracking cronológico comprobará la primera restricción, pero deberá esperar hasta que tratemos de instanciar x_4 antes de comprobar la segunda restricción.

El backtracking cronológico es un algoritmo muy simple pero es muy ineficiente. El problema es que tiene una visión local del problema. Sólo comprueba restricciones que están formadas por la variable actual y las pasadas, e ignora la relación entre la variable actual y las futuras. Además, este algoritmo es ingenuo en el sentido de que no 'recuerda' las acciones previas, y como resultado, puede repetir la misma acción varias veces innecesariamente. Para ayudar a combatir este problema, se han desarrollado algunos algoritmos de búsqueda más robustos. Estos algoritmos se pueden dividir en algoritmos *look-back* y *look-ahead*, que describiremos a continuación.

7.2. Algoritmos Look-Back

Los algoritmos look-back 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-back llevan a cabo la comprobación de la consistencia *hacia atrás*, es decir, entre la variable actual y las pasadas. Veamos algunos algoritmos look-back.

7.2.1. Algoritmos look-back

Backjumping(BJ) [20] es un algoritmo para CSPs 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 más profunda (más cerca de la variable actual) x_j que esta 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) [31] 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 [17] 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 $(x_1, a_1), ..., (x_{i-1}, a_{i-1}))$ 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.

7.3. Algoritmos look-ahead: Forward Checking

Como ya indicamos anteriormente, los algoritmos look-back tratan de reforzar el comportamiento de BT mediante un comportamiento más inteligente cuando se encuentran en situaciones sin salida. Sin embargo, todos ellos todavía llevan a cabo la comprobación de la consistencia solamente hacia atrás, ignorando las futuras variables. Los algoritmos *Look-ahead* hacen una comprobación hacia adelante en cada etapa de la búsqueda, es decir, ellos llevan a cabo las comprobaciones para obtener las inconsistencias de las variables futuras involucradas además de las variables actual y pasadas. De esa manera, las situaciones sin salida se pueden identificar antes y además los valores

inconsistentes se pueden descubrir y podar para las variables futuras.

7.3.1. Forward Checking

Forward checking (FC) [23] es uno de los algoritmos look-ahead más comunes. En cada etapa de la búsqueda, FC comprueba hacia adelante la asignación 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. 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 se puede ver como aplicar un simple paso de arco-consistencia sobre cada restricción que involucra la variable actual con una variable futura después de cada asignación de variable. A continuación presentamos el pseudo código de forward checking.

Proceso Forward-checking

1. Seleccionar x_i.
2. Instanciar $x_i \leftarrow a_i : a_i \in D_i$.
3. Razonar hacia adelante (check-forward):

   Eliminar de los dominios de las variables, aún no instanciadas con un valor, aquellos valores inconsistente con respecto a la instanciación $x_i \leftarrow a_i$, 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 $x_i \leftarrow a_i$:
   - 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.

Forward checking se ha combinado con algoritmos look-back para generar algoritmos *híbridos* [31]. Por ejemplo, *forward checking* con *conflict-directed backjumping* (FC-CBJ) [31] es un algoritmo híbrido que combina el movimiento hacia adelante de FC con el movimiento hacia atrás de CBJ, y de esa manera tiene las ventajas de ambos algoritmos.

Minimal forward checking (MFC) [11] es una version de FC que retrasa llevar a cabo todo la comprobación de la consistencia de FC hasta que es absolutamente necesario. En vez de comprobar hacia adelante la asignación actual contra todas las variables futuras, MFC sólo comprueba si la asignación actual causa una limpieza de dominios. Para hacer esto es suficiente comprobar la asignación actual con los valores de cada variable futura hasta que se encuentra una que es consistente. Después, si el algoritmo ha retrocedido, vuelve atrás y lleva a cabo las comprobaciones 'perdidas'. 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 % [11].

Recientemente Bacchus ha propuesto nuevas versiones de forward checking [1]. Estas versiones están basadas en la idea de desarrollar un mecanismo de poda de dominios que elimina valores no sólo en el nivel actual de búsqueda, sino que también en cualquier otro nivel. Bacchus primero describe una plantilla genérica para implementar varias versiones de forward checking y después describe cuatro instanciaciones de esa plantilla devolviendo nuevos algoritmos de forward checking. Los primeros dos algoritmos, *extended forward checking* (EFC) y EFC-, tienen la habilidad de podar futuros valores que son inconsistentes con asignaciones hechas antes de la asignación actual pero que no habían sido descubiertas. Los otros dos algoritmos, *conflict based forward checking* (CFFC) y CFFC-, están basados en la

idea de los conflictos, al igual que CBJ y learning para reforzar a FC. CFFC y CFFC- almacenan los conjuntos conflicto y los usan para podar lo 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.

8. Heurísticas

Un algoritmo de búsqueda para la satisfacción de restricciones requiere el orden en el cual se van a estudiar las variables, así como el orden en el que se van a instanciar los valores de cada una de las variables. Seleccionar el orden correcto de las variables y de los valores puede mejorar notablemente la eficiencia de resolución. También puede resultar importante una ordenación adecuada de las restricciones del problema [34]. Veamos las más importantes heurísticas de ordenación de variables y de ordenación de valores.

8.1. Ordenación de Variables

Experimentos y análisis de muchos investigadores han demostrado que el orden en el cual las variables son asignadas durante la búsqueda puede tener una impacto significativo en el tamaño del espacio de búsqueda explorado. 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 y dinámica.

- Las heurísticas de *ordenación de variables estáticas* generan un orden fijo de las variables antes de iniciar la búsqueda, basado en información global derivada del grafo de restricciones inicial.
- Las heurísticas de *ordenación de variables dinámicas* pueden cambiar el orden de las variables dinámicamente basándose en información local que se genera durante la búsqueda.

8.1.1. Heurísticas de ordenación de variables estáticas

En la literatura se han propuesto varias heurísticas de ordenación de variables estáticas. Estas heurísticas se basan en la información global que se deriva de la topología del grafo de restricciones original que representa el CSP.

- La heurística *minimum width* (MW) [15] impone en primer lugar un orden total sobre las variables, de forma que el orden tiene la mínima anchura, y entonces selecciona las variables en base a ese orden. La *anchura* de la variable x es el número de variables que están antes de x, de acuerdo a un orden dado, y que son adyacentes a x. La anchura de un orden es la máxima anchura de todas las variables bajo ese orden. La anchura de un grafo de restricciones es la anchura mínima de todos los posibles ordenes. Después de calcular la anchura de un grafo de restricciones, las variables se ordenan desde la última hasta la primera en anchura decreciente. Esto significa que las variables que están al principio de la ordenación son las más restringidas y las variables que están al final de la ordenación son las menos restringidas. Asignando las variables más restringidas al principio, las situaciones sin salida se pueden identificar antes y además se reduce el número de vueltas atrás.
- La heurística *maximun degree* (MD) [9] ordena las variables en un orden decreciente de su grado en el grafo de restricciones. El *grado* de un nodo se define como el número de nodos que son adyacentes a él. Esta heurística también tiene como objetivo encontrar un orden de anchura mínima, aunque no lo garantiza.
- La heurística *maximun cardinality* (MC) [32] selecciona la primera variable arbitrariamente y después en cada paso, selecciona la variable que es adyacente al conjunto más grande de las variables ya seleccionadas.

En [9] se compararon varias heurísticas de ordenación de variables estáticas utilizando CSPs generados aleatoriamente. Los resultados experimentales probaron que todos ellos son peores que el algoritmo *minimum remaining values* (MRV), que es una heurística de ordenación de variables dinámicas que presentaremos a continuación.

8.1.2. Heurísticas de ordenación de variables dinámicas

El problema de los algoritmos de ordenación estáticos es que ellos no tienen en cuenta los cambios en los dominios de las variables causados por la propagación de las restricciones durante la búsqueda, o por la densidad de las restricciones. Esto es porque estos heurísticas generalmente se utilizan en algoritmos de comprobación hacia atrás donde no se lleva a cabo la propagación de restricciones. Se han propuesto varias heurísticas de ordenación de variables dinámicas que abordan este problema.

La heurística de ordenación de variables dinámicas más común se basa en el principio de *primer fallo* (FF) [23] 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 con el principio de FF, en cada paso, seleccionaríamos la variable más restringida. La heurística FF también conocida como heurística *minimum remaining values* (MRV), trata de hacer lo mismo seleccionando la variable con el dominio más pequeño. Esto se basa en la intuición de que si una variable tiene pocos valores, entonces es más difícil encontrar un valor consistente. Cuando se utiliza MRV junto con algoritmos de comprobación hacia atrás, equivale a una heurística estática que ordena las variables de forma ascendente con la talla del dominio antes de llevar a cabo la búsqueda. Cuando MRV se utiliza en conjunción con algoritmos forward-checking, la ordenación se vuelve dinámica, ya que los valores de las futuras variables se pueden podar después de cada asignación de variables. En cada etapa de la búsqueda, la próxima variable a asignarse es la variable con el dominio más pequeño.

8.2. Ordenación de Valores

En comparación, se ha realizado poco trabajo sobre heurísticas para la ordenación de valores. La idea básica que hay detrás de las heurísticas de ordenación de valores 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.

Una de las heurísticas de ordenación de valores más conocidas es la heurística *min-conflicts* [28]. Básicamente, esta heurística ordena los valores de acuerdo a los conflictos en los que éstos están involucrados con las variables 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 que son incompatibles con a. El valor seleccionado es el asociado a la suma más baja. Esta heurística se puede generalizar para CSPs no binarios de forma directa. Cada valor a de la variable x_i se asocia con el número total de tuplas que son incompatibles con a en las restricciones en las que está involucrada la variable x_i. De nuevo se selecciona el valor con la menor suma. En [24] Keng y Yun proponen una variación de la idea anterior. De acuerdo a su heurística, cuando se cuenta el número de valores incompatibles para una futura variable x_k, éste 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 que actualmente estamos examinando. De nuevo los porcentajes se añaden para todas las variables futuras y se selecciona el valor más bajo que se obtiene en todas las sumas.

Geelen propuso una heurística de ordenación de valores a la cual llamó *promise* [21]. Para cada valor a de la variable x contamos el número de valores que soporta a en cada variable adyacente futura, y toma el producto de las cantidades contadas. Este producto se llama la promesa de un valor. De esta manera se selecciona el valor con la máxima promesa. Usando el producto en vez de la suma de los valores soporte, la heurística de Geelen 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. 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.

En [18] se describen tres heurísticas de ordenación de valores dinámicos inspirados por la intuición de que un subproblema es más probable que tenga solución si no tiene variables que tengan un sólo valor en su dominio.

- La primera heurística, llamada heurística *max-domain-size* selecciona el valor de la variable actual que crea el máximo dominio mínimo en las variables futuras.
- La segunda heurística, llamada *weighted-*

max-domain-size es una mejora de la primera. Esta heurística especifica una manera de romper empates basada en el número de futuras variables que tiene una talla de dominio dado. Por ejemplo, si un valor a_i deja cinco variables futuras con dominios de dos elementos, y otro valor a_j deja siete variables futuras también con dominios de dos elementos, en este caso se selecciona el valor a_i.

- La tercera heurística, llamada *point-domain-size*, que 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 uno debido a la variable a_i, se añaden 8 unidades al peso de a_i. De esta manera se selecciona el valor con el menor peso.

9. Otras técnicas de resolución de CSPs: Métodos estocásticos

Por métodos estocásticos no referimos a las técnicas de búsqueda no sistemáticas e incompletas incluyendo heurísticas. Ejemplos de este tipo de técnicas son *hill climbing* [3], *búsqueda tabú* [19], *enfriamiento simulado* y *algoritmos genéticos*. 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 alcanza la solución (con un cierto número de iteraciones). Estos métodos son generalmente robustos y buenos para encontrar un mínimo global en espacios de búsqueda grandes y complejos. En [35], Thornton presenta el éxitoso uso de algoritmos genéticos y técnicas de enfriamiento simulado para resolver grandes sistemas de inigualdades no lineales. El nombre *estocástico* indica, que estos métodos tienen un aspecto aleatorio que reduce la oportunidad de converger al mínimo local. Sin embargo, pueden ocurrir situaciones donde el proceso de búsqueda se encuentra en porciones erróneas del espacio de soluciones. Esto generalmente requiere que el sistema reinicie su ejecución empezando en otro punto de partida aleatorio. Un estudio más detallado puede verse en [25].

10. Aplicaciones

La programación de restricciones se ha aplicado con mucho éxito a muchos problemas de áreas tan diversas como planificación, scheduling, generación de horarios, empaquetamiento, diseño y configuración, diagnosis, modelado, recuperación de información, CAD/CAM, criptografía, etc.

Los problemas de asignación fueron quizás el primer tipo de aplicación industrial que fue resuelta con herramientas de restricciones. Entre los ejemplos típicos iniciales figuran la asignación de stands en los aeropuertos, donde los aviones deben aparcar en un stand disponible durante la estancia en el aeropuerto (aeropuerto Roissy en Paris) [12] o la asignación de pasillos de salida en el aeropuerto internacional de Hong Kong [7]. Otro ejemplo es la asignación de atracaderos en en el puerto internacional de Hong Kong [30].

Otra área de aplicación de restricciones típica es la asignación de personal donde las reglas de trabajo y unas regulaciones imponen una serie de restricciones difíciles de satisfacer. El principal objetivo en este tipo de problemas es balancear el trabajo entre las personas contratadas de manera que todos tengas las mismas ventajas. Existen sistemas como Gymnaste que se desarrolló para la generación de turnos de enfermeras en los hospitales [6], para la asignación de tripulación a los vuelos (British Airways, Swissair), la asignación de plantillas en compañías ferroviarias (SNCF, Compañía de Ferrocarril Italiana) [13] o sistemas para la obtención de mallas ferroviarias óptimas [2].

Sin embargo, una de las áreas de aplicación más exitosa de los CSPs con dominios finitos es en los problemas de secuenciación o Scheduling, donde de nuevo las restricciones expresan las limitaciones existentes en la vida real. El software basado en restricciones se utiliza para la secuenciación de actividades industriales, forestales, militares, etc. En general, el uso de las restricciones en sistemas complejos de planificación y scheduling se está incrementando cada vez más debido a las tendencias de las empresas de trabajar bajo demanda.

11. Tendencias

Las dos cuestiones básicas con respecto a la metodología CSP son (i) la modelización del problema

y (ii) la resolución de las restricciones. En cuanto a la modelización de problema es importante contar con un modelo de restricciones que capte correctamente el problema. En cuanto a la resolución del problema (obtención de soluciones) es importante contar con métodos eficientes y adecuados para la inherente complejidad NP del problema.

Debido a la gran cantidad de áreas donde se aplican la programación de restricciones, éstos se tienen que adaptar a la topología de los problemas que deben manejar. Hay problemas en los que hay pocas variables pero con restricciones muy complejas, otros con muchas variables y restricciones binarias, y en la mayoría de los casos son problemas con muchas variables, con grandes dominios y muchas restricciones.

Las tendencias actuales en la metodología CSP pasan por desarrollar técnicas para resolver determinados problemas de satisfacción de restricciones basándose principalmente en la topología de estos. Entre todas podemos destacar las siguientes:

- Técnicas para resolver problemas sobrerestringidos, donde la investigación se centran en el desarrollo de técnicas para resolver problemas con restricciones *hard* o requeridas y restricciones *soft* u opcionales.
- Técnicas para resolver problemas con muchas variables y restricciones, donde puede resultar conveniente la paralelización o distribución del problema de forma que éste se pueda dividir en un conjunto de subproblemas mas fáciles de manejar.
- Técnicas combinadas de satisfacción de restricciones con métodos de investigación operativa con el fin de obtener las ventajas de ambas propuestas.
- Técnicas de computación evolutiva, donde los algoritmos genéticos al igual que las redes neuronales, se han ido ganando poco a poco el reconocimiento de los investigadores como técnicas efectivas en problemas de gran complejidad. Se distinguen por utilizar estrategias distintas a la mayor parte de las técnicas de búsqueda clásicas y por usar operadores probabilísticos más robustos que los operadores determinísticos.
- Otros temas de interés son heurísticas inspiradas biológicamente, tales como la colonia de hormigas; la optimización mediante cúmulo de partículas; los algoritmos meméticos, los algoritmos culturales y la búsqueda dispersa, etc.

En esta monografía se incluyen artículos que tratarán algunas de las cuestiones anteriormente planteadas.

12. Conclusiones

En este trabajo hemos presentado los modelos y técnicas más relevantes de un problema de satisfacción de restricciones. Además hemos introducido la notación más importante que vamos a utilizar a lo largo de esta monografía. En ella, hemos hecho especial hincapié en las restricciones del CSP, ya que dependiendo de ellas, los problema adoptan distintas propiedades y por consiguiente distintas formas de resolverlos. También hemos presentado algunos de los niveles de consistencia existentes en la literatura cuyo objetivo principal es reducir el espacio de búsqueda. En el siguiente trabajo, se amplían estas técnicas y se profundiza aun más en la inferencia y búsqueda en CSPs. Por otra parte, los CSPs temporales [10] son un tipo especial de CSP donde las variables representan primitivas temporales (principalmente asociadas a puntos de tiempo, intervalos o duraciones). Las restricciones en este tipo de problema son restricciones temporales entre estas primitivas. Diversos trabajos presentados en esta monografía tratan específicamente sobre CSPs temporales y son especialmente importantes en dominios de problemas dinámicos en los que el tiempo juega un papel crucial. Otros artículos de la monografía tratan sobre restricciones geométricas, de especial relevancia en problemas de modelado físico y CAD/CAM. Adicionalmente también se incluyen trabajo sobre tipos específicos de CSPs tales como CSPs flexibles, borrosos, no binarios, con restricciones lógicas, etc. Además se incluyen trabajos en los que se presentan la aplicación práctica de los CSPs a problemas tales como la diagnosis, la recuperación de información y el acceso a bases de datos.

13. Agradecimientos

Este trabajo ha sido parcialmente financiado por el proyecto DPI2001-2094-C03-03 del MCYT y el

proyecto CTIDIB/2002/112 de la Generalitat Valenciana.

Referencias

[1] F. Bacchus. Extending forward checking. *In Proc. of the Sixth International Conference on Principles and Practice of Constraint Programming (CP2000)*, pages 35–51, 2000.

[2] F. Barber, M.A. Salido, Ingolotti L., M. Abril, A. Lova, and P. Tormos. An interactive train scheduling tool for solving and plotting running maps. *To appear in LNCS/LNAI on Artificial Intelligence Technology Transfer*, 2004.

[3] R. Barták. *Heuristic and stochastic algorithms.* http://ktilinux.ms.mff.cuni.cz/ bartak/constraints/stochastic.html, 1998.

[4] R. Barták. Constraint programming: In pursuit of the holy grail. *in Proceedings of WDS99 (invited lecture), Prague, June*, 1999.

[5] C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. *In Proc. Principles and Practice of Constraint Programming (CP-99)*, pages 88–102, 1999.

[6] P. Chan, K. Heus, and G. Weil. Nurse scheduling with global constraints in CHIP: GYMNASTE. *In Proc. of Practical Application of Constraint Technology*, 1998.

[7] K.P. Chow and M. Perett. Airport counter allocation using constraint logic programming. *In Proc. of Practical Application of Constraint Technology*, 1997.

[8] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. *In proceedings of the 15th IJCAI*, pages 412–417, 1997.

[9] R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraints satisfaction problems. *Artificial Intelligence*, 68:211–241, 1994.

[10] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint network. *Artificial Intelligence*, 49:61–95, 1991.

[11] M.J. Dent and R.E. Mercer. Minimal forward checking. *In Proc. of the 6th International Conference on Tools with Artificial Intelligence*, pages 432–438, 1994.

[12] M. Dincbas and H. Simosis. APACHE- a constraint based, automated stand allocation systems. *In Proc. of Advanced Software Technology in Air Transport*, 1991.

[13] F. Focacci, E. Lamma, P. Melo, and M. Milano. Constraint logic programming for the crew rostering problem. *In Proc. of Practical Application of Constraint Technology*, 1997.

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

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

[16] E. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM, 32*, 4:755–761, 1985.

[17] D. Frost and R. Dechter. Dead-end driven learning. *In Proc. of the National Conference on Artificial Intelligence*, pages 294–300, 1994.

[18] D. Frost and R. Dechter. Look-ahead value orderings for constraint satisfaction problems. *In Proc. of IJCAI-95*, pages 572–578, 1995.

[19] P. Galinier and J.K. Hao. Tabu search for maximal constraint satisfaction problems. *In Proceedings of International Conference on Principles and Practice of Constraint Programming (CP97)*, pages 196–208, 1997.

[20] J. Gaschnig. *Performance measurement and analysis of certain search algorithms.* Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[21] P.A. Geelen. Dual viewpoint heuristic for binary constraint satisfaction problems. *In proceeding of European Conference of Artificial Intelligence (ECAI'92)*, pages 31–35, 1992.

[22] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[23] R. Haralick and Elliot G. Increasing tree efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.

[24] N. Keng and D. Yun. A planning/scheduling methodology for the constrained resources problem. *In Proceeding of IJCAI-89*, pages 999–1003, 1989.

[25] J. Larrosa and P. Meseguer. Algoritmos para Satisfacción de Restricciones. *Inteligencia Artificial: Revista Iberoamericana de Inteligencia Artificial*, 20:31–42, 2003.

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

[27] F. Manya and C. Gomes. Técnicas de resolucion de problemas de satisfacción de restricciones. *Inteligencia Artificial: Revista Iberoamericana de Inteligencia Artificial*, 19:168–180, 2003.

[28] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. A heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[29] U. Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[30] M. Perett. Using constraint logic programming techniques in container port planning. *In ICL Technical Journal*, pages 537–545, 1991.

[31] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1993.

[32] P.W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[33] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. *In proceeding of European Conference of Artificial Intelligence (ECAI-94)*, pages 125–129, 1994.

[34] M.A. Salido and F. Barber. A constraint ordering heuristic for scheduling problems. *In Proceeding of the 1st Multidisciplinary International Conference on Scheduling : Theory and Applications*, pages 476–490, 2003.

[35] A.C. Thornton. Constraint specification and satisfaction in embodiment design. *PhD thesis, Department of Engineering, University of Cambridge, UK*, 1993.

[36] P. van Beek. On the minimality and decomposability of constraint networks. *In Proc. of the National Conference on Artificial Intelligence (AAAI-92)*, pages 447–452, 1999.

[37] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.