UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

MODELOS Y TÉCNICAS DE CONSISTENCIA EN PROBLEMAS DE SATISFACCIÓN DE RESTRICCIONES

Autor: Marlene Alicia Arangú Lobig
Director: Dr. D. Miguel Ángel Salido Gregorio

PARA LA OBTENCIÓN DEL GRADO DE
DOCTOR EN INFORMÁTICA
POR LA
UNIVERSITAT POLITÈCNICA DE VALÈNCIA
Valencia, España
DICIEMBRE 2011

A mi familia.

Tabla de Contenidos

Índice de figuras

Agradecimientos

En primer lugar quisiera agradecer a Miguel Ángel Salido, mi director, por su guía, paciencia y apoyo. En especial quiero agradecerle sus frases: *esto tiene buena pinta ...* y *no te preocupes, prueba con ...*, las cuales fueron un gran estímulo para seguir y alcanzar esta meta.

A los miembros del *Grupo de Investigación de Inteligencia Artificial, Planificación y Scheduling* de la Universidad Politécnica de Valencia, muy especialmente a su director Federico Barber, quien me inició en los CSPs, a sus explicaciones en el laboratorio y a que siempre me ha facilitado las cosas. A mis compañeros de grupo: Laura C., Laura I., Mariamar, Mario, y Montse: el investigar con ustedes, su amistad, su ayuda y los buenos ratos pasados juntos no los olvidaré.

A los miembros del GTI de la UPV, quienes me han tratado como una más de su grupo. En especial quiero agradecer a Eva, Toni y Oscarin, con quienes me inicié en la investigación en planificación en IA.

Por supuesto, quiero agradecer a mis familiares y amigos. Muy especialmente con quienes he compartido y vivido este sueño: mi esposo Jorge, y mis hijos: Alberto, Edu y Jorge L., a los que les pedí minutos que se transformaron en horas y más, por su paciencia, comprensión y amor. A mis padres, hermanos, tíos, suegros, cuñados y compadres, que me han brindado apoyo incondicional cada vez que lo he requerido. A mi familia en Valencia, España: Jorge Miró y familia, Sor Juliana, Laura I., Mariamar, Montse, Stella, Fanny y Luis B.: su amistad desinteresada, su apoyo y palabras de estímulo en todo momento, hicieron que me sintiera en casa.

A las autoridades de la Universidad Centroccidental "Lisandro Alvarado", especialmente a: Rosario C., Lulú, Celeste A., Francisco G., Fernando S. y Juan F., así como mis compañeros de trabajo: Laura S., Jose L., Lilia L. y Maura; por el apoyo que recibí para y durante la realización de mis estudios doctorales.

Por último, quiero agradecer a todas aquellas personas que directamente o indirectamente me han ayudado a llevar a cabo este trabajo.

Marlene Arangú Lobig

Resumen

Hoy en día, muchos problemas reales pueden ser modelados como problemas de satisfacción de restricciones (CSPs) y ser resueltos utilizando técnicas de programación de restricciones. Estos problemas pertenecen a diferentes áreas de conocimiento tales como inteligencia artificial, investigación operativa, sistemas de información, bases de datos, etc. Los CSPs pertenecen a la categoría de problemas NP-completos por lo que es necesario el uso de herramientas que simplifiquen el problema de forma que se encuentre una solución de forma eficiente. Las técnicas más comunes para manejar un CSP son las técnicas de consistencia y las técnicas de búsqueda. Las técnicas de consistencia o de filtrado tienen por objeto reducir el espacio de búsqueda mediante el filtrado de los dominios de las variables. Las técnicas de arco-consistencia son las más utilizadas ya que llevan a cabo una importante poda de dominios; eliminando valores que no formaran parte de la solución; de una manera eficiente.

Por ello, proponer algoritmos que alcancen la arco-consistencia ha sido un punto central en la comunidad científica reduciendo la complejidad tanto espacial como temporal de estos algoritmos.

Sin embargo, muchos trabajos que investigan la arco-consistencia llevan a cabo asunciones que no están presentes en muchos problemas de la vida real. Por ejemplo, un mismo par de variables puede participar en más de una restricción, lo que se denomina problema no-normalizado, y cuando el tamaño de los dominios es grande, el proceso de normalización puede resultar prohibitivo. En estos casos la 2-consistencia lleva a cabo una reducción de los dominios de las variables mayor que la arco-consistencia por lo que los algoritmos de búsqueda pueden resultar más

eficientes. No obstante, la literatura es escasa en relación al desarrollo de algoritmos que alcancen la 2-consistencia.

En esta tesis presentamos nuevos algoritmos de arco-consistencia y de 2-consistencia como algoritmos de preproceso para la resolución de problemas de satisfacción de restricciones. Estos algoritmos presentan un mejor comportamiento que las actuales técnicas existentes en la literatura en algunos de los criterios de eficiencia establecidos por la comunidad científica: reducción del número de chequeos de restricciones, reducción de la cantidad de propagaciones, disminución del tiempo de cómputo, o el aumento de la cantidad de poda. En este trabajo también presentamos un algoritmo de normalización híbrido, que permite calcular el coste del proceso de normalización requerido para aplicar las técnicas existentes en la literatura. En cuanto a las técnicas de búsqueda, en esta tesis presentamos dos algoritmos: un algoritmo heurístico de propósito general y un algoritmo completo y dependiente del dominio. Ambos algoritmos se benefician de los resultados proporcionados por las técnicas de 2-consistencia desarrolladas en esta tesis.

Adicionalmente y debido a que muchos problemas reales pueden modelarse de forma natural mediante el uso de restricciones no-binarias, en esta tesis proponemos un lenguaje de modelado que combina el uso de CSPs con restricciones no-binarias y planificación POCL para problemas de planificación y scheduling. El lenguaje de modelado propuesto permite codificar restricciones complejas, expresar el uso de los recursos continuos y beneficiarse de las técnicas de consistencia desarrolladas en esta tesis.

Todas las técnicas presentadas se han evaluado empíricamente sobre diferentes instancias de problemas aleatorios y benchmarks. También han sido aplicadas a la resolución del problema de planificación de horarios ferroviarios.

Abstract

Nowadays, many real problems can be modeled as constraint satisfaction problems (CSPs) and be solved using constraint programming techniques. These problems belong to different areas of knowledge such as: Artificial Intelligence, Operations Research, Information Systems, databases, etc. CSPs are NP-complete, so it is necessary to use tools that simplify the problem in order to find a solution in an efficient way. The most common techniques to handle a CSP are consistency techniques and search techniques. Consistency techniques, or filtering techniques, are designed to reduce the search space by filtering the variable domains. The arc-consistency techniques are mostly used since they carry out significant pruning of domains; by efficiently removing values that will not take part of a solution.

Therefore, proposing efficient algorithms to achieve arc-consistency has always been considered as a central question in the scientific community.

However, many works on arc-consistency make assumptions that are not present in many real life problems. For instance, the same pair of variables may be involved in two or more constraints, which is called non-normalized problem. Therefore, these problems would require a normalization process in order to use such algorithms, which can be prohibitive in large domains. In these cases, the 2-consistency is able to prune more search space than arc-consistency, so that search algorithms can be more efficient. However, little works have focused their attention in the development of 2-consistency algorithms.

In this thesis, we present new algorithms for arc-consistency and 2-consistency as preprocessing algorithms to solve constraint satisfaction problems. These algorithms

have a better performance than the state of the art techniques in some efficiency criteria established by the scientific community, such as: number of constraints checks, amount of propagation, CPU time, or amount of pruning. Furthermore, we introduce a hybrid normalization algorithm, which calculates the cost of the normalization process required to apply the techniques that have been published in the literature.

Regarding search techniques, we present two algorithms: a general purpose heuristic search algorithm and a complete domain dependent search algorithm. Both algorithms take benefit from the results given by the developed 2-consistency techniques.

Additionally, since many real problems can be modeled naturally by using non-binary constraints, in this thesis we propose a modeling language that combines CSPs with non-binary constraints and POCL planning for planning and scheduling problems. The proposed modeling language enables to encode complex constraints, to represent the continuous use of resources and to benefit from the consistency techniques developed in this thesis.

All the presented techniques have been empirically evaluated on different random instances and benchmarks. They have also been applied to solve the railway scheduling problem.

Resùm

Hui en dia, molts problemes reals poden ser modelats com a problemes de satisfacció de restriccions (CSPs) i ser resolts utilitzant tècniques de programació de restriccions. Estos problemes pertanyen a diferents àrees de coneixement com ara la intel·ligència artificial, la investigació operativa, els sistemes d'informació, les bases de dades, etc. Els CSPs pertanyen a la categoria de problemes NP-complets pel que és necessari l'ús de ferramentes que simplifiquen el problema de manera que es trobe una solució de forma eficient. Les tècniques més comuns per a manejar un CSP són les tècniques de consistència i les tècniques de recerca. Les tècniques de consistència o de filtrat tenen com a objecte reduir l'espai de recerca mitjançant el filtrat dels dominis de les variables. Les tècniques d'arc-consistència són les més utilitzades ja que duen a terme una important poda de dominis, eliminant valors que no formen part de la solució, d'una manera eficient.

Per això, proposar algorismes que aconseguixen l'arc-consistència ha sigut un punt central en la comunitat científica per a reduir la complexitat tant espacial com temporal d'aquestos algorismes.

No obstant això, molts treballs que investiguen l'arc-consistència duen a terme assumpcions que no estan presents en molts problemes de la vida real. Per exemple, una mateixa parella de variables poden participar en més d'una restricció, el que es denomina problema no-normalitzat, i quan la grandària dels dominis és gran, el procés de normalització pot resultar prohibitiu. En estos casos la 2-consistència du a terme una reducció dels dominis de les variables major que l'arc-consistència, pel que els algorismes de recerca poden resultar més eficients. No obstant això, la literatura és escassa en relació al desenvolupament d'algorismes que aconseguixen

la 2-consistencia.

En aquesta tesi presentem nous algorismes d'arc-consistència i de 2-consistència com algorismes de preprocés per a la resolució de problemes de satisfacció de restriccions. Estos algorismes presenten un millor comportament que les actuals tècniques existents a la literatura en alguns dels criteris d'eficiència establerts per la comunitat científica: reducció del nombre de revisions de restriccions, reducció de la quantitat de propagacions, disminució del temps de còmput, o l'augment de la quantitat de poda. En aquest treball també presentem un algorisme de normalització híbrid, que permet calcular el cost del procés de normalització requerit per a aplicar les tècniques existents a la literatura. En quant a les tècniques de recerca, en aquesta tesi presentem dos algorismes: un algorisme heurístic de propòsit general i un algorisme complet i dependent del domini. Ambdós algorismes es beneficien dels resultats proporcionats per les tècniques de 2-consistència desenvolupades en aquesta tesi.

Addicionalment, i pel fet que molts problemes reals poden modelar-se de forma natural per mitjà de l'ús de restriccions no-binàries, en aquesta tesi proposem un llenguatge de modelatge que combina l'ús de CSPs amb restriccions no-binàries i planificació POCL per a problemes de planificació i scheduling. El llenguatge de modelatge proposat permet codificar restriccions complexes, expressar l'ús dels recursos continus i beneficiar-se de les tècniques de consistència desenvolupades en aquesta tesi.

Totes les tècniques presentades s'han avaluat empíricament amb diferents instàncies de problemes aleatoris i benchmarks. També han sigut aplicades a la resolució del problema de planificació d'horaris ferroviaris.

Capítulo 1

Introducción

1.1. Generalidades

Las restricciones están presentes en nuestro día a día: cantidad de memoria en nuestro ordenador, número de asientos en el coche, cantidad de dinero disponible para hacer la compra, etc. Así, una restricción es una limitación en un espacio de posibilidades. Formular problemas en términos de restricciones permite una formulación declarativa natural de *qué* debe ser satisfecho, sin decir *cómo* debe ser satisfecho [41]. Así mismo, esta formulación ha demostrado ser útil para el modelado de actividades tales como: scheduling, planificación, diagnosis, razonamiento temporal y espacial, así como en tareas de ingeniería y en modelos biológicos.

Si el número restricciones es pequeño y además el conjunto de posibilidades a escoger es reducido, el problema se puede resolver fácilmente, sin necesidad de algoritmos sofisticados que ayuden a realizar el cómputo. Sin embargo, cuando la complejidad del problema crece, ya sea por el número de restricciones o valores posibles que pueden combinarse, se hace necesario el uso de tecnologías eficientes que permitan encontrar una solución al problema en un tiempo razonable.

La programación de restricciones es el estudio de los sistemas computacionales basados en restricciones, y es una tecnología de software emergente para la descripción y resolución eficaz de problemas de optimización combinatoria en problemas reales [25, 26]. Esta investigación está enfocada a una de las áreas de la programación de restricciones: la satisfacción de restricciones, específicamente en las técnicas de inferencia local que permiten reducir el espacio de búsqueda de los Problemas de

Satisfacción de Restricciones o CSPs (de las siglas en inglés Constraint Satisfaction Problems).

1.2. Problemas de Satisfacción de Restricciones

Los problemas de satisfacción de restricciones (CSPs) son unas estructuras útiles para la resolución de diversos tipos de problemas y son ampliamente utilizados en el área de Inteligencia Artificial (IA) [23, 41, 22, 88]. Un CSP consiste en un conjunto finito de variables, cada una de las cuales posee un dominio de valores y existen un conjunto de restricciones que acotan la combinación de valores que las variables pueden tomar. Una solución para un CSP es una asignación de un valor del dominio de cada variable de forma que satisfaga todas las restricciones del problema.

Modelar un problema como un CSP confiere ventajas importantes, como son: (a) la representación de los estados se ajusta a un modelo estándar (compuesto por variables, dominios y restricciones, donde se pueden asignar valores a las variables); (b) pueden desarrollarse heurísticas genéricas eficaces que no requieran ninguna información dependiente del problema y (c) la estructura del grafo de restricciones puede ser utilizada para simplificar el proceso de solución en problemas binarios [102] (en un grafo de restricciones los nodos corresponden a las variables y las aristas corresponden a las restricciones). Si al CSP se le añade una función objetivo, este se transforma en un CSOP o COP (de las siglas en inglés Constraint Satisfaction and Optimization Problem, o Constraint Optimization Problem).

La resolución de un CSP consta de dos fases diferentes [26]:

- *Fase de modelado:* consiste en expresar el problema en términos de la sintaxis de los CSP: variables, dominios y restricciones.

- *Fase de solución:* consiste en aplicar técnicas de satisfacción de restricciones para resolver el CSP, las cuales pueden clasificarse en: técnicas completas, técnicas incompletas (o de inferencia) y técnicas híbridas.

Generalmente la declaración de los problemas viene expresada en lenguaje natural. Para poder trabajar estos problemas utilizando técnicas CSP, se requiere

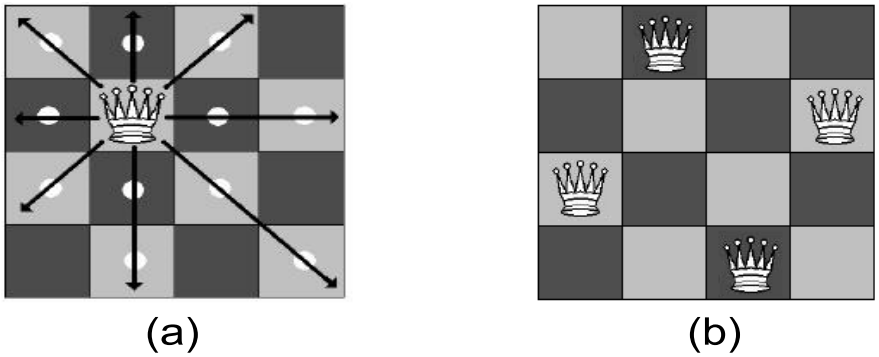


Figura 1.1: Problema de 4-reinas: (a) movimientos posibles que puede realizar una reina y (b) una solución al problema.

modelar el problema en términos de: variables, dominios y restricciones. Las restricciones pueden ser representadas *intensionalmente*, mediante una ecuación o *extensionalmente*, mediante su conjunto de tuplas válidas o no válidas[1]. Así mismo, las restricciones pueden estar compuestas por una, dos o más variables. En función del número de variables que componen la restricción hablamos de restricción unaria (una variable), restricción binaria (dos variables) y restricción no-binaria (tres o más variables). Una adecuada modelización del problema hará que éste sea resuelto de forma más eficiente.

Ejemplo. El problema de las n-reinas es un problema clásico en los CSPs: se trata de un acertijo que consiste en colocar n reinas en un tablero de ajedrez de dimensiones $n \times n$ de forma que no se amenacen entre ellas. En el juego de ajedrez, la reina amenaza a todas aquellas fichas que se encuentren en su misma fila, columna o diagonal. Los posibles movimientos que puede realizar una reina son mostrados en la Figura 1.1 (izquierda).

Una forma de representar este problema consiste en modelizar las filas del tablero como variables y las columnas como valores del dominio. Así, cuando decimos que asignamos el valor a a la variable X_i (con $1 \leq a \leq n$ y $1 \leq i \leq n$) estamos diciendo que se ha colocado la reina X_i en la intersección de la fila i con la columna a.

[1]Tupla: es un elemento del producto cartesiano de los dominios de las variables que participan en la restricción. En el próximo capítulo será definida formalmente.

Para representar que dos reinas no pueden atacarse entre sí, se debe representar que las reinas no pueden colocarse ni en la misma columna ni en la misma diagonal. Para ello, a cada par de variables X_i y X_j se le asigna la siguiente restricción: $R_{ij} = \{(a, b) \mid a \neq b \wedge \mid i - j \mid \neq \mid a - b \mid\} (i > j)$.

De esta forma, la modelización del problema de 4 reinas como un CSP es la siguiente:

- Variables: $X = \{X_1, X_2, X_3, X_4\}$, una variable por reina.
- Dominios: $D_i = \{1, 2, 3, 4\}$, un valor por columna.
- Restricciones: $\forall X_i, X_j$ con $1 \leq i, j \leq 4$ y $i \neq j$:

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

Tras aplicar una técnica de búsqueda al CSP modelado, una solución (de las dos posibles) al problema de 4-reinas es mostrada en la Figura 1.1 (b), donde la primera reina es colocada en la columna 2, la segunda es colocada en la columna 4, la tercera es colocada en la columna 1 y la cuarta reina es colocada en la columna 3 del tablero de ajedrez.

1.3. Búsqueda e inferencia en CSPs

Una técnica de solución para los CSPs son los algoritmos de búsqueda sistemática, los cuales tienen como base la búsqueda con retrocesos (backtracking). A grandes rasgos, el proceso de búsqueda de una solución utilizando la técnica de búsqueda con backtracking se realiza de la siguiente forma: (1) se escoge una variable, (2) se le asigna a dicha variable el primer valor de su dominio (se instancia la variable); (3) se verifica que el valor asignado sea compatible con los valores ya asignados anteriormente, según las restricciones del problema. (4) De no ser válida la instanciación con una restricción, la variable es instanciada con su siguiente valor del dominio y se realiza la verificación (se vuelve al paso (3)). (5) Si se agota el dominio de valores en la última variable instanciada, el algoritmo *retrocede* (hace backtrack) a una variable anterior y vuelve al paso (4). Esta secuencia de pasos será repetida hasta: (a) encontrar una combinación de valores para todas las variables que satisfaga todas

las restricciones del problema o (b) agotar el dominio de valores de alguna de las variables. En el caso (a) se ha encontrado una solución al problema y en el caso (b) se ha determinado que el problema no tiene solución.

La búsqueda sistemática es un método bastante sencillo, sin embargo, sufre con frecuencia de una explosión combinatoria del 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 se enfrentan los algoritmos de búsqueda sistemática es la continua aparición de inconsistencias locales [83].

Las inconsistencias locales son instanciaciones $\langle variable, valor \rangle$ que no satisfacen la restricción que las contiene, y por lo tanto no pueden formar parte de la solución. Por ejemplo, si la instanciación $\langle X, a \rangle$ es incompatible con todos los valores del dominio de la variable Y que está ligada a la variable X mediante una restricción, entonces el valor a es inconsistente con respecto a la propiedad local llamada 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, ya que estas instanciaciones inconsistentes no formaran parte de ninguna solución. De la misma forma, puede haber valores que son consistentes con respecto a la propiedad A pero son inconsistentes con respecto a otra consistencia local B. A los procesos A y B los llamaremos consistencia *local*. Otra forma de realizar el proceso de consistencia puede ser el denominado consistencia *global*, donde todos los valores que no pueden participar en una solución son eliminados. Pero, este nivel de consistencia (*global*) puede ser más costoso que el proceso de búsqueda de una solución, por lo que la consistencia *local* es la que se utiliza en la práctica.

Las restricciones explícitas de un CSP, cuando se combinan, generan algunas restricciones implícitas que pueden causar inconsistencias locales. Si un algoritmo de búsqueda no almacena dichas inconsistencias locales, repetidamente intentará instanciarlas, malgastando así tiempo de búsqueda.

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: $X \neq Y$ y $Y < Z$. Si asumimos que la búsqueda con backtracking trata de instanciar las

[2]La arco-consistencia será tratada en la siguiente sub-sección

variables en el orden X, Y y Z, entonces el algoritmo probará todas las posibles combinaciones de valores para las variables X, Y, Z antes de descubrir que no existe solución alguna. Si observamos la restricción existente entre las variables Y y Z, podremos verificar que no hay ninguna combinación de valores para ambas variables que satisfaga la restricción. Si el algoritmo pudiera identificar esta *inconsistencia local* antes, se evitaría un gran esfuerzo de búsqueda.

Así, dependiendo del número de variables involucradas en la restricción, se pueden conseguir diferentes niveles de consistencia: nodo-consistencia (una variable); arco-consistencia (dos variables); senda-consistencia (tres variables) y k-consistencia (k-variables). Sin embargo, la arco-consistencia es la técnica más utilizada [79, 28, 41, 24].

1.3.1. Arco-consistencia

La arco-consistencia es una técnica de inferencia incompleta que se aplica a restricciones donde participan dos variables. Se denomina *arco* a la restricción que vincula a las dos variables. También es la técnica más utilizada de propagar restricciones en dominios discretos. Se fundamenta en una observación realizada por Fikes [47] que expresa: dado dos dominios discretos D_i y D_j, para dos variables X_i y X_j las cuales son nodo consistentes, si existe un valor $a \in D_i$ y no existe un valor $b \in D_j$ que satisfaga una restricción R_{ij}, entonces el valor a puede ser eliminado de D_i. Esto puede hacerse debido a que el valor a no participará en ninguna solución del problema.

Hay que tener en cuenta que la idea de arco-consistencia propuesta en [47] es direccional, es decir, si un arco $(X_i \rightarrow X_j)$ es consistente, no significa que el arco inverso $(X_j \rightarrow X_i)$ sea también consistente (donde el arco $(X_i \rightarrow X_j) \equiv R_{ij}$ y $(X_j \rightarrow X_i) \equiv R_{ji}$).

Ejemplo. Consideremos el grafo que se muestra en la Figura 1.2 que consta de dos variables X_i y X_j, con dominios $D_i = \{1, 2, 3\}$ y $D_j = \{-1, 0, 1, 2, 3, 4\}$ y la restricción $R_{ij} : X_i \leq X_j$. Si realizamos el proceso de arco-consistencia en sentido $i \rightarrow j$ (ver Figura 1.3 (a)) todos los valores de D_i son *consistentes* ya que tienen uno o más valores en D_j (soportes) que satisfacen la restricción R_{ij}; los valores $X_j = -1$ y $X_j = 0$ son *inconsistentes* al no existir valores en D_i que satisfagan la restricción

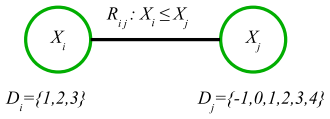


Figura 1.2: Ejemplo de CSP con dos variables X_i y X_j y una restricción R_{ij}.

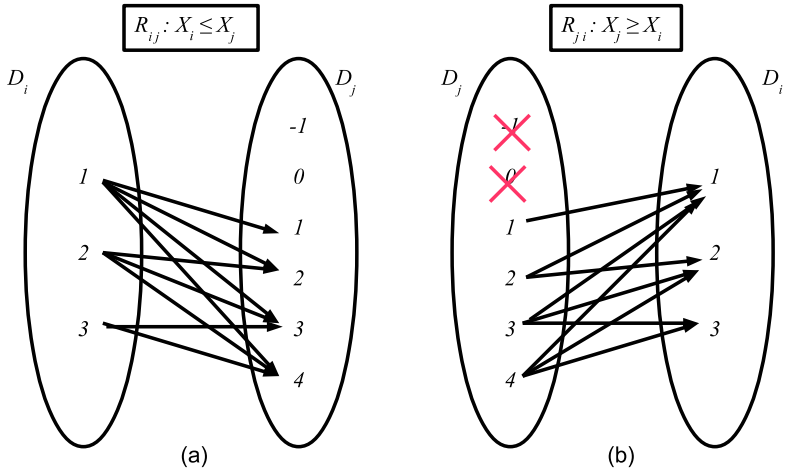


Figura 1.3: Proceso de arco-consistencia realizado al CSP del ejemplo 1.2. Cada flecha relaciona un valor con un soporte encontrado.

R_{ji} y pueden ser eliminados (podados) (ver Figura 1.3 (b)). Así, con el proceso de revisión en sentido $X_i \rightarrow X_j$ no se han podado valores del dominio de X_j, esta poda ha ocurrido al realizar la revisión en sentido $X_j \rightarrow X_i$. A este proceso de revisión de arcos (restricciones) en ambos sentidos se le denomina *arco-consistencia completa* o simplemente *arco-consistencia*.

Cuando la arco-consistencia elimina un valor del dominio de una variable, extiende o propaga dicha poda en el CSP. La forma de propagar en la arco-consistencia, es llamada *granularidad* [28]. Así, se habla de *granularidad gruesa* (coarse-grained) cuando se realiza la propagación a nivel de restricciones y de *granularidad fina* (fine-grained) cuando la propagación se realiza a nivel de los valores del dominio de cada una de las variables.

Una vez se ha alcanzado la arco-consistencia en un CSP, se puede dar cualquiera de estas situaciones:

1. Si el dominio de cada una de las variables contiene un único valor, se ha encontrado una solución al problema.

2. Si el dominio de una de las variables se queda vacío, entonces no existe solución al problema.

3. En cualquier otro caso, es necesario utilizar una técnica de búsqueda que determine si el CSP tiene o no solución.

Observación: El hecho de que un CSP sea arco consistente no significa que tenga al menos una solución. El ejemplo que se muestra en la Figura 1.4, es un CSP arco-consistente pero sin solución.

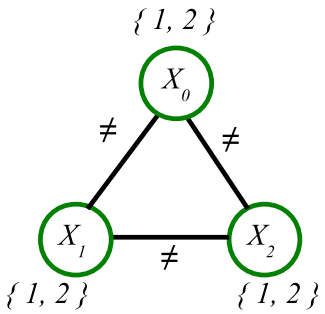


Figura 1.4: Ejemplo de CSP arco-consistente sin solución.

Se han propuesto una variedad de algoritmos que establecen la arco-consistencia tales como: AC1, AC2 y AC3 [83]; AC4 [89]; AC5 [93]; AC6 [29]; AC7 [30]; AC8 [39]; AC2001/3.1 [33]; AC3.2 y AC3.3 [76] AC3rm [77]; AC3^{bit} [79]; L, L', L'' [120]; etc. A continuación proporcionamos una breve explicación de los algoritmos de arco-consistencia antes mencionados:

Los algoritmos AC1, AC2, AC3 [83] revisan repetidamente el dominio de las variables para eliminar los valores no consistentes (que no tienen soportes). Debido a su sencillez, son fáciles de implementar y consumen muy pocos recursos. Sin embargo, ellos efectúan muchos chequeos inefectivos.

El algoritmo AC4 [89] es el único algoritmo que confirma la existencia de un soporte no identificándolo a través de un proceso búsqueda. Para ello, el algoritmo almacena todos los soportes de cada valor en estructuras de datos auxiliares. Es un algoritmo óptimo. Su ineficiencia radica en su complejidad espacial y la necesidad de mantener estructuras de datos muy grandes. El algoritmo AC5 [93, 71] permite realizar una arco-consistencia especializada para restricciones funcionales (cualquiera de los valores de los dominios de las variables que participan en la restricción, satisfacen dicha restricción, ejemplo $X = Y + 5$), restricciones anti-funcionales (negación de la restricción funcional, ejemplo $X \neq Y$) y restricciones monotónicas (requieren que exista un orden total en el dominio de forma tal que cuando los valores a y b satisfacen una restricción R, los valores a' y b' también satisfacen dicha restricción R, donde $a' \leq a \wedge b' \geq b$. Ejemplo: $X \leq Y - 3$).

El algoritmo AC6 [29] mantiene estructuras de datos más ligeras que AC4. En efecto, la idea en AC6 no es buscar todos los soportes que tiene un valor en una restricción, sino asegurar que al menos tiene uno. El algoritmo AC7 [30] mejora la idea de los valores de soporte aplicadas en AC4 y AC6. El algoritmo AC7 utiliza el meta-conocimiento que infiere de los soportes para aplicar la bidireccionalidad y evitar chequeos. La idea es que si un valor $a \in D_i$ soporta a un valor $b \in D_j$ entonces b también soporta a a. El algoritmo AC8 [39] está basado en soportes, pero sin guardar información sobre ellos. AC8 trabaja de forma similar a AC3 pero en vez de almacenar restricciones almacena las referencias de las variables en una lista y mantiene su status en un vector.

El algoritmo AC2001/3.1 [33] sigue el mismo marco que AC3, requiere de un orden total ascendente en los dominios de las variables y almacena el soporte más pequeño de cada valor en la restricción en un puntero. AC2001/3.1 difiere de AC3 por su procedimiento de revisión y por su fase de inicialización.

Los algoritmos AC3.2 y AC3.3 [76] están basados en AC2001/3.1. Utilizan la bidireccionalidad parcial y total, para lo cual añaden nuevas estructuras: un vector para almacenar si es válida la tupla (restricción, variable, valor) y un vector que almacena el número de soportes externos válidos para cada una de las tuplas. Al igual que AC2001/3.1, AC3.2 y AC3.3 modifican el procedimiento de revisión para evaluar las estructuras anteriores antes de efectuar nuevos chequeos de restricciones.

Los algoritmos AC3r y AC3rm [77] refinan al algoritmo AC3 almacenando *residuos de los soportes* (soporte que ha sido almacenado durante la ejecución previa de la revisión, el cual determina si un valor es soportado por una restricción). Así, antes de buscar un soporte, es verificada la validez del residuo asociado con el valor. La diferencia entre AC3r y AC3rm es que en este último es explotada la bidireccionalidad.

Existen muchos algoritmos de arco-consistencia, sin embargo AC3 y AC4 son los más utilizados [24] y en esta década la investigación se ha centrado en refinar los algoritmos AC3 [33, 79, 120] y AC2001/3.1 [76, 77]. En el próximo capítulo se realizará una explicación más detallada de estos algoritmos.

1.4. Problemas de Planificación y Scheduling. Modelización vía CSP

En Inteligencia Artificial un problema real de planificación va íntimamente ligado a uno de scheduling, donde la Planificación tiene como propósito seleccionar un conjunto de acciones cuya ejecución permita, partiendo de una situación inicial determinada, lograr alcanzar una situación objetivo y el Scheduling busca resolver el problema de asignar recursos, sujetos a restricciones, a un conjunto de tareas (o acciones) [41, 72].

Un problema de Planificación y Scheduling ($P_{P\&S}$) puede ser modelado como un CSP [117, 86, 98, 121, 27, 26], donde las variables representarán a las acciones y las restricciones modelarán las limitaciones del problema, así como los mecanismos de razonamiento que permitan la elección de las acciones para su resolución. Sin embargo, los problemas de $P_{P\&S}$ modelados como CSP son NP-completos y su optimización los convierte en problemas NP-hard. Para resolver eficientemente este tipo de problemas, se requieren buenos algoritmos de búsqueda y filtrado.

En el proceso de planificación, cada acción a ser seleccionada puede requerir de pre-condiciones que deben ser ciertas en el momento que las requiera la acción. La aplicación de dicha acción produce efectos que permitirán satisfacer las pre-condiciones de otras acciones o bien permitirá que se alcance la situación objetivo

[68, 100, 102]. Así mismo, en situaciones reales muchos recursos son limitados y pueden haber restricciones (sobre los recursos, sobre las acciones o sobre los objetivos a ser alcanzados) que deben ser satisfechos para que el plan resultante sea válido. La asignación de recursos y la consideración de restricciones son aspectos que pueden ser resueltos eficientemente, mediante técnicas de Scheduling [88]. Por ejemplo, para un problema de administración de terminales de contenedores marítimos, las restricciones de Scheduling podrían ser: cantidad de recursos limitada (hay sólo dos grúas y tres embarcaderos); restricciones de precedencia (primero se carga del patio de contenedores 1 y luego del patio de contenedores 2), plazos máximos de finalización (las mercancías deben estar en los barcos antes de las 16:00 horas).

Planificación y scheduling (P&S) son líneas de investigación que a menudo se solapan [115, 91, 54, 2, 60]. La resolución de un problema de este tipo consiste en la obtención de un plan viable, es decir que pueda ejecutarse con las restricciones impuestas por el problema. En general un problema de P&S ($P_{P\&S}$) se traduce en un problema de búsqueda en un espacio complejo que posee restricciones y una función objetivo que guía la optimización, lo que implica un problema combinatorio en cuanto al número de alternativas que pueden conducir a una de las posibles soluciones.

En la resolución de un problema de $P_{P\&S}$ podemos distinguir tres etapas: 1) comienza con el *modelado del problema*, utilizando para ello un lenguaje, que permita representar de forma fácil y completa el problema a resolver; 2) continúa con la *planificación*, donde se obtiene un plan de acciones sintáctica y semánticamente correctos y, 3) finaliza con el *scheduling*: donde se fijan las acciones en el tiempo, se comprueba la viabilidad del plan en relación a los recursos disponibles y se optimiza la solución, según la función objetivo.

Para la tipología de problemas $P_{P\&S}$, nos centraremos en la etapa 1: específicamente en el enriquecimiento del *lenguaje de modelado*. A tal efecto, la comunidad de planificación tiene un lenguaje estándar llamado PDDL (del inglés Planning Domain Definition Language) con diferentes versiones (1.2, 2.1, 2.2 y 3.0) [81, 45, 64]. A pesar de que la semántica de PDDL es bastante amplia para modelar problemas, no contempla ciertas características presentes en problemas reales, como por ejemplo: a) la representación de acciones y restricciones complejas y b) el modelado de recursos continuos, sin que ello signifique la adición de nuevas acciones. En el

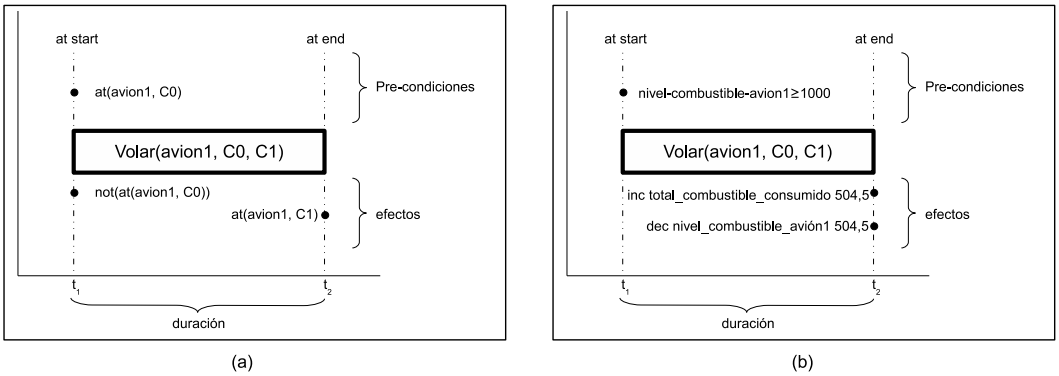


Figura 1.5: Representación del modelo conservativo de acciones con PDDL2.1 para la Acción Volar: (a) sólo variables proposicionales; (b) sólo variables numéricas.

caso de la comunidad de scheduling, la mayoría de resolutores utiliza un lenguaje de modelado propietario que maximiza sus capacidades, sin que se logre un acuerdo de lenguaje de modelado estándar entre los mismos.

Tomando como base el lenguaje PDDL y la propuesta de modelado para problemas de $P_{P\&S}$ realizada por Vidal y Geffner en CPT [121], (equivalente al PDDL2.1), observamos que la duración de las acciones es un intervalo de tiempo definido, las pre-condiciones son requeridas al inicio de la acción (at start) y los efectos son generados cuando finaliza la acción (at end). Esto se conoce como modelo conservativo de acciones.

Ejemplo. Sea una acción que consiste en desplazarse entre dos ciudades $C0$ y $C1$ con un avión $avion1$, que consume 504.5 unidades de combustible. A esta acción la llamaremos $Volar$. Una representación de la acción $Volar$, utilizando el modelo conservativo de acciones se hace combinando los incisos a) y b) de la Figura 1.5. En la Figura 1.5 (a) son representadas sus variables (pre-condiciones y efectos) proposicionales y en la Figura 1.5 (b) son representadas sus variables numéricas.

No obstante, el modelado de problemas utilizando el modelo conservativo de acciones tiene limitaciones como: que no permite que las condiciones y los efectos sean requeridos y/o generados en cualquier instante de tiempo mientras transcurre la acción, ni tampoco lo permite fuera del intervalo de duración de la misma (antes o después que se inicie/finalice la acción); no puede expresar la utilización y/o requerimientos de recursos continuos y la métrica se limita a minimizar la duración

total del plan. Adicionalmente, tampoco proporciona flexibilidad para incorporar un conjunto de acciones (o plan de entrada) que permita diferentes modos de trabajo en el resolutor. Si se desea representar situaciones más reales, como por ejemplo, la condición de que el avión $avion1$ debe estar en la ciudad $C0$ cinco unidades de tiempo antes de que se inicie la acción $Volar$ (el desplazamiento del avión hacia la pista de despegue), o se desea representar el efecto del desplazamiento de dicho avión durante diez unidades de tiempo por la pista cuando se dirige a la terminal (antes del desembarque), es necesario con el modelo conservativo agregar nuevas acciones como *Despegue* y *Aterrizaje*, que serán instanciadas para cada avión, lo que incrementa el espacio de búsqueda. Adicionalmente, si ya se conoce de antemano que cierto conjunto de acciones deben ser ejecutadas (de forma obligatoria, a petición de uno de los directivos de la terminal) dichos requerimientos (un plan parcial de entrada) no pueden ser expresados en el modelo conservativo. Tampoco podemos representar una métrica que sea minimizar la cantidad de combustible.

Por lo antes expuesto y debido a que los CSP facilitan el modelado de restricciones, resulta interesante enriquecer el lenguaje de modelado, utilizando el marco de los CSPs y así ser capaz de representar las situaciones previamente descritas, presentes en problemas de $P_{P\&S}$.

1.5. Motivación y Contribución

Los problemas de satisfacción de restricciones y los problemas de optimización son computacionalmente difíciles de resolver ya que pertenecen a la categoría de los problemas NP-completos y NP-duros, respectivamente. Instancias de estos problemas de tamaño realista son a menudo demasiado complejos de resolver y las aproximaciones existentes no son adecuadas para resolverlas en un tiempo aceptable. Por ello, la obtención de la solución óptima en instancias concretas de problemas reales requiere el desarrollo de nuevos métodos de relajación del problema, mejorados a su vez con aproximaciones heurísticas. Una alternativa importante, cuando es computacionalmente difícil obtener la solución óptima, es el desarrollo de técnicas de aproximación garantizando la bondad de las mismas.

Uno de los puntos claves de la satisfacción de restricciones es usar activamente las restricciones para eliminar valores inviables de los dominios de las variables y

consecuentemente podar el espacio de búsqueda. Es lo que se conoce como Técnicas de Inferencia, Técnicas de Propagación de Restricciones o Técnicas de Consistencia. Dichas técnicas son una parte esencial de los resolutores de CSPs ya que los beneficios de la propagación son inmediatos.

Los algoritmos que fuerzan la consistencia trabajan con una solución parcial de una sub-red de restricciones más pequeña y dicha solución la van extendiendo al resto de la red de restricciones. El número posible de combinaciones puede ser enorme, mientras que pocos valores pueden ser consistentes. Eliminando valores superfluos de la definición del problema, el tamaño del espacio de soluciones decrece. Si un dominio se torna vacío como resultado de dicha reducción, se sabe inmediatamente que el problema no tiene solución [103].

Los algoritmos de arco-consistencia están basados en la noción de soporte. Estos algoritmos aseguran que cada valor en el dominio de cada variable está soportado por un valor de la otra variable que participa en la restricción. Proponer algoritmos eficientes que alcancen la arco-consistencia ha sido un punto central en la comunidad que investiga el razonamiento con restricciones, ya que la arco-consistencia es el mecanismo básico de propagación utilizado en la mayoría de resolutores CSPs académicos e industriales [87]; ésta consume un porcentaje considerable del tiempo que se requiere para encontrar la solución del problema [120] y que las nuevas ideas que permiten mejorar la eficiencia de los algoritmos de arco-consistencia se puedan extender a otros algoritmos que alcancen otros niveles de consistencia local [101]. Por estas razones, la reducción de la complejidad espacial y temporal de los algoritmos de arco-consistencia es un área abierta aunque ampliamente investigada.

Estudiando y analizando en detalle los algoritmos de arco-consistencia anteriormente mencionados, hemos detectado ineficiencias, tales como:

- Cuando se podan valores de los dominios, se activa el proceso de propagación. Analizando este proceso hemos detectado (caso de AC3 extrapolable a AC2001/3.1, AC3.2, AC3.3 y AC3rm) que en cierto tipo de restricciones (orden), su re-evaluación no genera ninguna poda. En el caso de AC4, también hemos detectado que propaga variables a re-evaluación innecesariamente, ya que no son soporte de otras variables.

- La sub-utilización de estructuras en el caso de AC4, al no almacenar simétricamente el soporte encontrado.
- La ausencia de bidireccionalidad en algoritmos base, como en el caso de AC4.
- AC7 no funciona correctamente en entornos de problemas binarios no-normalizados, donde puede existir más de una restricción entre un mismo par de variables.

Así mismo, debido a que las restricciones no-binarias pueden ser convertidas en restricciones binarias utilizando diversos métodos como por ejemplo el método del grafo dual (the dual graph method) y el método de la variable oculta (the hidden variable method) [24, 90, 106], la mayoría de los algoritmos de consistencia asumen que los problemas tienen restricciones binarias. Adicionalmente, para simplificar la codificación y los conceptos a presentar, se asume que las restricciones son normalizadas (dos restricciones diferentes no involucran las mismas variables).

En [101], se menciona un extraño efecto de asociar arco-consistencia con 2-consistencia. La 2-consistencia garantiza que cualquier instanciación de un valor a una variable puede ser extendido a una segunda variable. En problemas binarios no-normalizados la 2-consistencia es más restrictiva que la arco-consistencia. Sólo en problemas binarios normalizados, arco-consistencia y 2-consistencia efectúan la misma poda. En general, esto no se cumple.

Ejemplo. La Figura 1.6 muestra en su parte izquierda el grafo de un CSP binario y no-normalizado (presentado en [101]) con dos variables X_1 y X_2, $D_1 = D_2 = \{1, 2, 3\}$ y dos restricciones $R_{12} : X_1 \leq X_2$, $R'_{12} : X_1 \neq X_2$. Como se puede observar, este CSP es arco-consistente ya que cada valor de cada variable tiene al menos un soporte para las restricciones R_{12} y R'_{12}. En este caso la arco-consistencia no poda ningún valor del dominio de las variables X_1 y X_2. Sin embargo, como se muestra en [101], el CSP no es 2-consistente debido a que la instanciación $X_1 = 3$ no puede ser extendida a X_2 y la instanciación $X_2 = 1$ no puede ser extendida a X_1. Así, la Figura 1.6 (derecha) presenta el CSP resultante filtrado por la arco-consistencia y la 2-consistencia.

Como puede observarse en el ejemplo anterior, la 2-consistencia es más restrictiva que la arco-consistencia. Sin embargo, no se han encontrado en la literatura algoritmos que lleven a cabo la 2-consistencia.

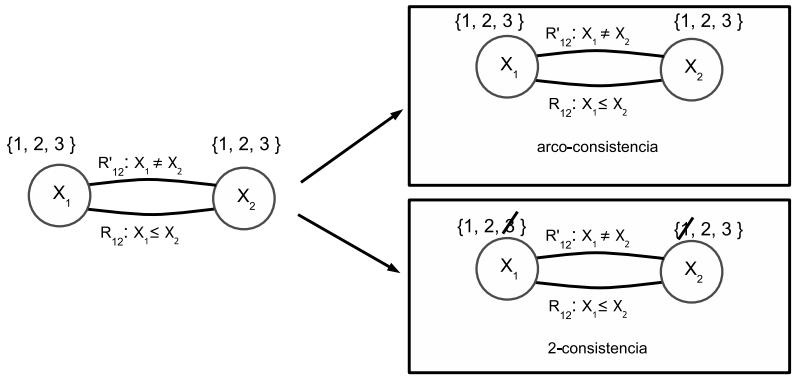


Figura 1.6: Ejemplo de CSP binario y no-normalizado y CSPs resultantes de al aplicar: arco-consistencia y 2-consistencia.

Por otra parte, en cuanto al modelado de los problemas de $P_{P\&S}$, además de las características propuestas por PDDL [81, 45, 64], el lenguaje de modelado debe permitir representar características presentes en problemas reales, lo cual puede hacerse mediante una codificación de CSPs no-binaria que contenga: (a) un modelo más elaborado de acciones que permita representar condiciones y efectos que no estén inmersas dentro de la duración de la acción, lo cual facilitará el modelado de situaciones reales sin necesidad de añadir nuevas acciones; (b) la representación de restricciones complejas (restricciones de precedencia, restricciones temporales, tiempos máximos de finalización, ventanas temporales) para acciones y proposiciones y (c) el manejo de recursos continuos (fluents) en condiciones y efectos.

El objetivo de esta tesis consiste en desarrollar modelos y técnicas de consistencia para resolver CSPs y está motivado por los siguientes puntos:

- la arco-consistencia es la técnica de filtrado más utilizada en los resolutores[25];
- se han detectado ineficiencias en los algoritmos de arco-consistencia;
- existe una confusión al asociar arco-consistencia con 2-consistencia [101];
- aunque se ha sugerido su extensión en [28], las ideas propuestas en los algoritmos de arco-consistencia no han sido extendidas para problemas binarios no-normalizados;

- se puede añadir bidireccionalidad a los algoritmos que no la tienen para mejorar su eficacia;
- se puede reducir la cantidad de propagaciones que realizan los algoritmos de consistencia, basándose en el tipo de restricciones y en las estructuras de datos que dichos algoritmos utilizan;
- se pueden re-utilizar los mecanismos de las diferentes técnicas de consistencia para realizar nuevas técnicas de consistencia local;
- se considera interesante medir empíricamente el coste del proceso de normalización, previo al proceso de la arco-consistencia, debido a las asunciones estándares de que los problemas son binarios y normalizados. Se obtendrán medidas de desempeño (cantidad de podas, número de chequeos de restricciones, cantidad de propagaciones y tiempo de cómputo) para contrastarlas con las obtenidas por otras técnicas de consistencia que no requieran el proceso de normalización.
- la necesidad de incrementar la expresividad de los lenguajes de modelado para solucionar problemas con restricciones y acciones complejas, utilizando una herramienta automatizada que sea flexible y que permita diferentes niveles de completitud de un plan de entrada (desde plan de entrada vacío hasta plan de acciones completo).

1.5.1. Contribuciones

Esta tesis considera la necesidad de obtener modelos y técnicas que permitan manejar de manera eficiente los CSPs. Se ha trabajado en las dos fases del CSP: a) modelado del problema como CSP y b) búsqueda de la solución. Por ello, las contribuciones de estas tesis se clasifican según el tipo de restricciones que existen en los problemas:

- Problemas binarios, cuyas restricciones contienen dos variables, donde hemos diseñado y reformulado técnicas de solución incompletas y completas. Específicamente, técnicas de consistencia y de técnicas búsqueda. Debido a que pretendemos abordar problemas reales, asumimos que las restricciones están representadas de forma intensional (mediante una función matemática). Para

esta tipología de problemas proponemos:

  - técnicas de consistencia: reformulación de algunas técnicas de arco-consistencia existentes y el diseño de nuevas técnicas de 2-consistencia que sean más eficientes, es decir, que realicen menos chequeos de restricciones, menos propagaciones, sean más rápidas (consuman menos tiempo de cómputo) y así como una mayor cantidad de podas;
  - técnicas de búsqueda: diseño de nuevas técnicas de búsqueda dependientes e independientes del dominio, que apoyadas en las técnicas de consistencia propuestas, permitan encontrar más eficientemente una solución.

- Problemas no-binarios, cuyas restricciones contienen un número arbitrario de variables, para los cuales proponemos la extensión de un lenguaje de modelado utilizando CSPs y planificación POCL para problemas de planificación y scheduling, que permita modelar acciones y restricciones complejas similares a las que se encuentran en los problemas reales.

1.5.2. Objetivos

El objetivo general de la tesis doctoral es contribuir al avance en el desarrollo de modelos y técnicas de resolución de problemas de programación con restricciones en el tratamiento de problemas complejos. En particular, se hizo énfasis en la formulación y en la resolución de problemas de satisfacción de restricciones (CSPs) mediante la mejora, desarrollo y aplicación de técnicas de consistencia. Así mismo, se estudió el impacto de las técnicas de consistencia propuestas en etapas de preproceso y durante la búsqueda, en entornos centralizados para problemas clásicos, problemas aleatorios y para un problema real como lo es la planificación de horarios en el transporte ferroviario.

Para alcanzar el objetivo general de la tesis doctoral, hemos desarrollado los siguientes objetivos específicos:

- Analizar problemas, modelos y métodos, para lo cual realizamos una revisión actualizada de técnicas para los problemas de satisfacción de restricciones, haciendo especial hincapié en las técnicas de inferencia local y técnicas de búsqueda aplicadas a los CSPs.

- Desarrollar técnicas de inferencia local para CSPs. Nos centramos en el desarrollo de técnicas novedosas de inferencia local que permitiran la resolución más eficiente de los CSPs normalizados y no-normalizados.
- Desarrollar técnicas de búsqueda para CSPs, mediante el desarrollo de técnicas novedosas de búsqueda heurística que hicieran uso de las técnicas de inferencia desarrolladas previamente.
- Desarrollar un lenguaje de modelado que permita la codificación de restricciones complejas.
- Aplicar las técnicas desarrolladas a escenarios de evaluación ampliamente reconocidos por la comunidad científica. Contrastar los resultados con los obtenidos a partir de las técnicas existentes.
- Aplicar y evaluar en un entorno real las técnicas de consistencia y búsqueda desarrolladas: resolución de problemas complejos en el contexto del trasporte ferroviario.

1.6. Estructura del Trabajo

El resto de la tesis está organizada como sigue:

- En el Capítulo 2, presentamos un breve resumen del estado del arte en los problemas de satisfacción de restricciones que nos permite identificar las líneas actuales de desarrollo. En este capítulo definimos los problemas de satisfacción de restricciones (CSP), sus componentes (variables, dominios y restricciones) y las distintas técnicas existentes para resolverlos, haciendo un especial énfasis en las técnicas de consistencia local.

Las aportaciones específicas de esta tesis se centran en los siguientes capítulos:

- En el Capítulo 3, presentamos los algoritmos de consistencia desarrollados que alcanzan la *arco-consistencia* en problemas binarios normalizados: AC3-OP [9, 11], AC4-OP [10] y AC2001-OP [15]; y en problemas binarios no-normalizados: AC3-NN [12] y AC4-OPNN [12, 6].

- En el Capítulo 4, presentamos los algoritmos de consistencia que se han desarrollado, los cuales alcanzan la *2-consistencia* en problemas binarios no-normalizados: 2-C3 [7], 2-C3-OP [8, 13], 2-C3OPL [16, 17], 2-C4 [19], 2-C6 y AC3NH [14].

- En el Capítulo 5, presentamos dos técnicas de búsqueda para CSPs: una técnica de búsqueda heurística incompleta (BLS), la cual es independiente del dominio y, una técnica de búsqueda completa dependiente del dominio (SchTrains) la cual ha sido aplicada al problema de asignación de horarios ferroviarios [18].

- En el Capítulo 6, presentamos la evaluación empírica de los algoritmos propuestos en los capítulos anteriores, tanto en problemas generados aleatoriamente como en problemas de referencia (benchmarks).

- En el Capítulo 7, presentamos la modelización binaria del problema de asignación de horarios ferroviarios, así como la evaluación de los algoritmos propuestos sobre problemas benchmarks de planificación ferroviaria.

- En el Capítulo 8, presentamos una codificación de CSPs no-binaria, la cual es una extensión al lenguaje de modelado de CTP [6]. Combina POCL y CSP y es capaz de modelar problemas reales con restricciones complejas (problemas que integran Planificación y Scheduling, $P_{P\&S}$). La codificación permite resolver los problemas de forma centralizada y distribuida, así como hacer reparaciones de planes. Así mismo, presentamos a ClassP, una arquitectura que permite el modelado y resolución de problemas de $P_{P\&S}$ en tres modos de trabajo: scheduler, pseudo-planner y planificador. También presentamos los resultados de la evaluación que ha sido realizada sobre diferentes grupos de problemas: problemas benchmarks de la Competición Internacional de Planificación (IPC), problemas Ad-hoc y un caso de aplicación al problema de transporte aéreo [59, 4, 55, 111, 112, 5, 3].

- Finalmente, en el Capítulo 9 presentamos las conclusiones, las publicaciones relacionadas con este trabajo y los trabajos futuros.

Capítulo 2

Problemas de Satisfacción de Restricciones: Estado del Arte

2.1. Generalidades

La programación de restricciones es una tecnología de software emergente para la descripción y resolución eficaz de problemas de optimización combinatoria en problemas reales [23]. La programación de restricciones estudia los sistemas computacionales basados en restricciones y abarca dos áreas: solución de restricciones y satisfacción de restricciones [25]. La *solución de restricciones* trata con problemas cuyos dominios son infinitos y con restricciones no lineales. Utiliza métodos algebraicos y numéricos para la resolución de los problemas. Por otro lado, la *satisfacción de restricciones* trabaja con problemas combinatorios donde existe un número finito de variables, dominios y restricciones. Esta tesis doctoral se enfoca en los problemas de satisfacción de restricciones (CSPs).

Los problemas de satisfacción de restricciones o CSPs consisten en un conjunto finito de variables, cada una de las cuales posee un dominio de valores y existen un conjunto de restricciones que acotan la combinación de valores que las variables pueden tomar. Así por ejemplo, un CSP con 10 variables, y cada variable con 10 posibles valores en su dominio, tendría un total de diez mil millones de posibilidades diferentes.

Dado que por lo general los CSP son problemas NP-completos, diversas técnicas han sido diseñadas para conseguir solución a este tipo de problemas, las cuales son agrupadas en: técnicas de búsqueda, técnicas de inferencia y técnicas híbridas

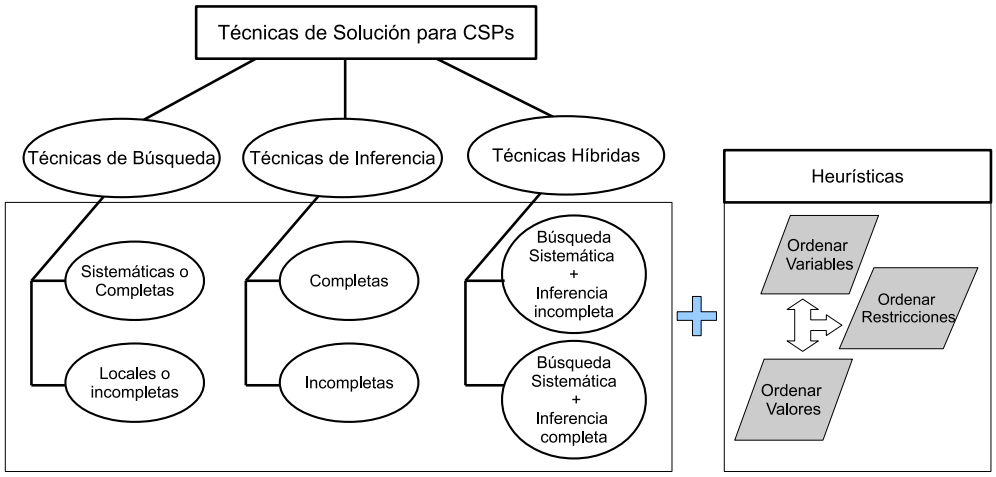


Figura 2.1: Técnicas de solución para los CSPs.

[75]. Las **técnicas de búsqueda** consisten en explorar el espacio de estados hasta encontrar una solución, y pueden ser sistemáticas (genera y prueba, backtracking cronológico) o locales (tabú search, genetic algorithm). Las **técnicas de inferencia** consisten en deducir un CSP' equivalente que sea más fácil de resolver. Estas técnicas pueden ser completas -si se logra extraer la solución del CSP' en forma directa- o incompletas -si se requiere complementarla con un proceso de búsqueda posterior, para encontrar la solución-. Ejemplo de inferencia incompleta son las técnicas de consistencia. **Las técnicas híbridas** consisten en combinar las dos técnicas anteriores. La Figura 2.1 proporciona un esquema global de las técnicas antes mencionadas.

Adicionalmente, para mejorar el desempeño de los algoritmos de solución en los CSP, se han agregado heurísticas como por ejemplo: heurísticas de ordenación de variables, de ordenación de valores y de ordenación de restricciones, lo cual ha generado algoritmos más eficientes para resolver los problemas. Así, en el presente capítulo se introducen los conceptos básicos relacionados con los problemas de satisfacción de restricciones (CSP) y las principales técnicas de solución encontradas en la literatura.

2.2. Definiciones

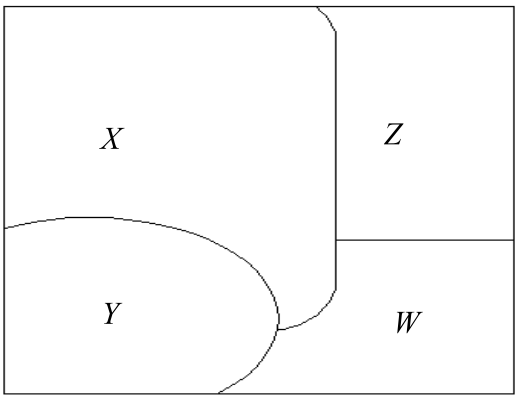
En esta sección resumiremos las notaciones y las definiciones básicas usadas en este trabajo, siguiendo los estándares propuestos en [28, 24, 41]:

Definición 2.2.1. Las *variables* son objetos o ítems los cuales pueden tomar diversos valores. El conjunto posible de valores que puede tomar la variable se denomina *dominio*. Las *restricciones* son reglas que imponen una limitación de valores a la variable, o a la combinación de variables, que debe ser asignada.

Definición 2.2.2. Un *Problema de Satisfacción de Restricciones*, está formado por la terna $P = \langle X, D, R \rangle$, donde:

- X es el conjunto finito de variables $\{X_1, X_2, ..., X_n\}$.
- D es el conjunto de dominios $D = \{D_1, D_2, ..., D_n\}$ tal que, para cada variable $X_i \in X$, existe un conjunto finito de valores $D_i \in D$ que la variable X_i puede tomar.
- R es el conjunto finito de restricciones $R = \{R_1, R_2, ..., R_m\}$, el cual restringe los valores que las variables pueden simultáneamente tomar.

Ejemplo. El problema de coloración de mapas es un problema clásico que se puede formular como un CSP. En este problema, se disponen de tres colores: rojo (r), verde (v), azul (a). Tenemos un mapa compuesto de cuatro regiones que deben ser coloreadas de manera tal que las regiones adyacentes tengan distintos colores. En la formulación del CSP, mostrada en la Figura 2.2, hay una variable por cada región del mapa. El dominio de cada variable es el conjunto de colores disponibles $\{r, v, a\}$. Las restricciones de este problema expresan que regiones adyacentes deben ser coloreadas con diferentes colores, por lo que se formulan cinco restricciones.



X={ X, Y, Z, W }

D={ D_X= [r, v, a], D_Y= [r, v, a], D_Z= [r, v, a], D_W= [r, v, a] }

R= { R_1: $X \neq Z$; R_2: $X \neq Y$; R_3: $X \neq W$; R_4: $Y \neq W$; R_5: $Z \neq W$}

Figura 2.2: Problema de coloración del mapa y su representación CSP.

Definición 2.2.3. Una *instanciación* es el par $\langle X_i, a\rangle$ que representa una asignación del valor a a la variable X_i, donde a pertenece al dominio de X_i ($a \in D_i$). Una instanciación de un conjunto de variables es una tupla de pares ordenados $\{\langle X_1, a_1\rangle, \langle X_2, a_2\rangle, ..., \langle X_i, a_i\rangle\}$, donde cada par ordenado $\langle X_i, a_i\rangle$ representa la asignación del valor de a_i a la variable X_i. A la tupla $\{\langle X_1, a_1\rangle, ..., \langle X_i, a_i\rangle\}$ que satisfaga la restricción que contiene sus variables se le denomina *tupla consistente*.

Definición 2.2.4. Una *solución* a un CSP es una instanciación de todas las variables de forma que se satisfagan todas las restricciones. Una *solución parcial* es una tupla consistente que contiene algunas de las variables del problema. Un problema es consistente, si al menos existe una tupla $\{\langle X_1, a_1\rangle, ..., \langle X_n, a_n\rangle\}$ consistente. En caso contrario diremos que el problema es inconsistente.

Así, para el ejemplo de Figura 2.2, una instanciación para la variable W podría ser $\langle W, a\rangle$ y una solución para el problema podría ser asignar los colores de la

siguiente forma: $\{\langle X, r\rangle, \langle Y, v\rangle, \langle Z, v\rangle, \langle W, a\rangle\}$. En esta asignación todas las variables correspondientes a regiones adyacentes tienen valores diferentes.

Básicamente, los objetivos que se desean alcanzar con un CSP se centran en determinar [22]:

- si el problema es consistente, es decir, si el problema tiene solución;
- una solución, sin preferencia alguna;
- todas las soluciones;
- la solución óptima, o al menos una buena solución, que optimice una función objetivo definida en términos de algunas o todas las variables.

Definición 2.2.5. Un CSP es *binario* si y solo si todas sus restricciones en R involucran dos variables X_i y X_j, con $i \neq j$.

Definición 2.2.6. Un CSP es *normalizado* si y solo si dos restricciones diferentes en R no involucran exactamente a las mismas variables.

A partir de la definición 2.2.2 pueden distinguirse varios tipos de CSPs, acorde al dominio de las variables, al número de variables por restricción o a las variables involucradas en las restricciones, como se muestra en la Figura 2.3. Así, un CSP, cuyas variables tengan dominios finitos (por ejemplo, colores: $azul, verde, rojo$, etc.) será un *CSP discreto*; un CSP, cuyas variables posean dominios continuos (por ejemplo, $[n, +\infty[$), será un *CSP continuo*; mientras que, un *CSP mixto*, estará formado por algunas variables con dominios discretos y otras variables con dominios continuos. Un CSP, cuyas restricciones estén formadas por una única variable, será un *CSP unario*; un CSP, que contenga restricciones donde participen dos variables, será un *CSP binario*; mientras que, un CSP, cuyas restricciones contengan tres o más variables, será un *CSP n-ario*. Un CSP, cuyas mismas variables no estén presentes en más de una restricción, será un *CSP normalizado* y si las mismas variables participan en dos o más restricciones, será un *CSP no-normalizado*.

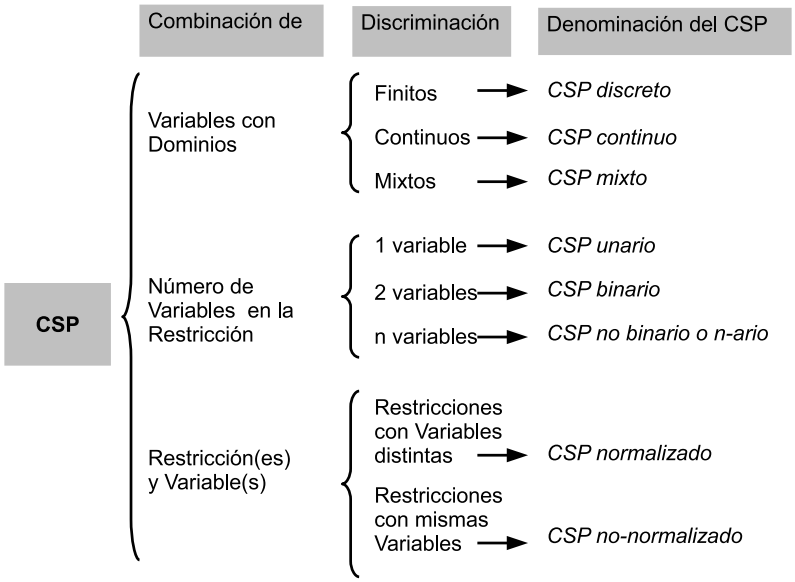


Figura 2.3: Esquema de clasificación para los CSPs.

La mayor parte de los trabajos de investigación que se realizan sobre problemas de satisfacción de restricciones, se hacen sobre problemas binarios normalizados, debido a que este tipo de restricciones se puede modelar fácilmente como un grafo de restricciones y porque así se simplifica la explicación de los algoritmos propuestos [28, 24, 41].

2.2.1. Notación

Antes de describir con mayor detalle los problemas de satisfacción de restricciones, vamos a presentar la notación que utilizaremos a lo largo de este trabajo.

El número de variables de un CSP lo denotaremos por n. La talla máxima del dominio del CSP la denotaremos con la letra d. La talla del dominio de una variable X_i la denotamos por d_i. El número de restricciones del CSP lo denotaremos por m.

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 números enteros o letras del alfabeto.

Dominios/Valores: El dominio de una variable X_i lo denotamos por D_i. A los valores individuales genéricos de un dominio los representaremos mediante las primeras letras del alfabeto: a, b, c, y al igual que en las variables también pueden ir seguidas de subíndices. La instanciación de un valor a a una variable X_i la denotaremos mediante el par $\langle X_i, a\rangle$. En esta tesis cuando hablemos de dominios, nos referiremos a dominios discretos, cuyos valores pertenecen a los números enteros.

Restricciones: Una restricción binaria entre dos variables X_i y X_j la denotaremos por R_{ij}. Una restricción $k-aria$ entre las variables $\{X_1, ..., X_k\}$ la denotaremos por $R_{1..k}$.

Las notaciones que surjan a lo largo del trabajo las definiremos cuando sean necesarias.

2.2.2. Restricciones

Vamos a dar algunas definiciones sobre las restricciones y explicaremos algunas de sus propiedades básicas.

Definición 2.2.7. La *aridad* de una restricción es el número de variables que componen dicha restricción. La denotaremos con la letra k.

Definición 2.2.8. Una *restricción unaria* es una restricción que contiene una sola variable. Sin pérdida de generalidad la notación que se empleará en las restricciones unarias será la siguiente:

$R_i : X_i \;\; op \;\; a$, donde $op \in \{<, \leq, >, \geq, =, \neq\}$ y $a \in \mathbb{Z}$

Definición 2.2.9. Una *restricción binaria* es una restricción que contiene dos variables. En la representación de restricciones binarias y sin pérdida de generalidad, utilizaremos indistintamente cualquiera de las siguientes notaciones:

- $R_{ij} : X_i \pm X_j \ \ op \ \ a$, donde $op \in \{<, \leq, >, \geq, =, \neq\}$ y $a \in \mathbb{Z}$
- $R_{ij} : X_i \ \ op \ \ \pm X_j \pm a$, donde $op \in \{<, \leq, >, \geq, =, \neq\}$ y $a \in \mathbb{Z}$

Definición 2.2.10. Una *restricción n-aria* o no-binaria es una restricción que involucra a un número arbitrario de n variables, con $n > 2$. Sin pérdida de generalidad la notación que se emplea en las restricciones n-arias es:

$R_{i,j..k} : X_i \pm X_j \pm ... \pm X_k \ \ op \ \ a$, donde $op \in \{<, \leq, >, \geq, =, \neq\}$; $k \leq n$ y $a \in \mathbb{Z}$

Ejemplos. La restricción $R_i : X_i \leq 5$ es una restricción unaria sobre la variable X_i. Las restricciones $R_{ij} : X_i - X_j \leq 3$ y $R_{ij} : X_i \neq X_j$ son restricciones binarias. La restricción $2X_1 - X_2 + 4X_3 \leq 4$ es una restricción no-binaria.

Definición 2.2.11. Una *tupla p* de una restricción $R_{i..k}$ es un elemento del producto cartesiano $D_i \times .. \times D_k$. Una tupla t que satisface la restricción $R_{i..k}$ se le llama *tupla válida* o permitida. Una tupla t que no satisface la restricción $R_{i..k}$ se le llama *tupla no válida* o tupla no permitida.

Así, al proceso de verificar si una tupla t es permitida o no por una restricción, se le llama chequeo de restricción, chequeo de la consistencia o comprobación de la consistencia. Este concepto se explicará con más detalle en los próximos capítulos.

Definición 2.2.12. Las restricciones R en un CSP pueden ser representadas *intensionalmente*, mediante una expresión algebraica o *extensionalmente*, mediante un conjunto de tuplas válidas o no válidas. Así, un CSP cuyas restricciones estén representadas en ambas formas (intensionalmente y extensionalmente) lo denominaremos *híbrido*.

Ejemplo. Consideremos una restricción entre dos variables X_1 y X_2, con dominios $D_1 = D_2 = \{0, 1, 2\}$, donde la variable X_1 es menor que la variable X_2. Esta restricción se representa de forma intensional mediante la ecuación: $R_{12} : X_1 < X_2$. La representación extensional de dicha restricción por medio de tuplas puede realizarse de dos formas distintas:

- tuplas permitidas (válidas): $R_{12} = \{(0,1),(0,2),(1,2)\}$
- tuplas no permitidas: $R_{12} = \{(0,0),(1,0),(1,1),(2,0),(2,1),(2,2)\}$

Ejemplo. Consideremos una restricción entre cuatro variables X_1, X_2, X_3, X_4 con dominios continuos [-1,3],[1,2],[1,2] y [2,5] respectivamente, donde la suma de las variables X_1, X_2 y X_3 es menor que la variable X_4. La representación intensional de esta restricción es: $R_{1234} : X_1 + X_2 + X_3 < X_4$. Sin embargo su representación extensional no es posible, ya que el número de combinaciones de tuplas válidas o no válidas es infinito, a menos que se asigne una granularidad.

Definición 2.2.13. Las restricciones R serán *normalizadas* si y solo si restricciones diferentes en R no involucran exactamente a las mismas variables. Si dos o más restricciones diferentes en R involucran a las mismas variables, las restricciones serán *no-normalizadas*.

Ejemplo. Sean las variables $X = \{X_1, X_2\}$ y con una única restricción entre ellas $R_{12} : X_1 \leq X_2$. R_{12} es una restricción normalizada. Sean las variables $X = \{X_3, X_4\}$ y las restricciones $R_{34} : X_3 \leq X_4$, $R_{34} : 2X_3 - X_4 \leq 3$. Ambas restricciones R_{34} son restricciones no-normalizadas.

Definición 2.2.14. *Simetría de la restricción*: si el valor $b \in D_j$ soporta al valor $a \in D_i$, entonces a también soporta a b.

2.2.3. Grafo de Restricciones

Los conceptos de la teoría de grafos juegan un papel fundamental en la modelización y resolución de los CSPs binarios ya que pueden ser representados como un grafo dirigido y ponderado, donde los nodos representan las variables y las aristas representan la restricciones existentes entre los nodos que forman dicha arista. Este grafo se llama grafo de restricciones o red de restricciones.

La representación de un CSP en un grafo de restricciones puede ser utilizada para simplificar el proceso de solución, produciendo en algunos casos una reducción

exponencial de la complejidad del problema. Una restricción unaria R_i es representada en el grafo como un bucle sobre el nodo i. Una restricción binaria R_{ij} entre las variables X_i y X_j, es representada con una arista que une al nodo i con el nodo j. La arista simétrica que une el nodo j con el nodo i correspondiente a la restricción R_{ji} (restricción inversa), por lo general es omitida, o es colocada en forma de líneas discontinuas. Cuando se quiere representar el sentido de una restricción, las aristas se sustituyen por flechas. Cuando existe más de una restricción entre un par de variables, se le agrega a la restricción el símbolo $'$, para indicar que es una restricción diferente a la anterior.

La representación de CSPs no-binarios se hace por medio de un hipergrafo donde los nodos representan las variables, y las hiperaristas representan las restricciones no-binarias [41]. Una hiperarista conecta los nodos que componen la correspondiente restricción no-binaria.

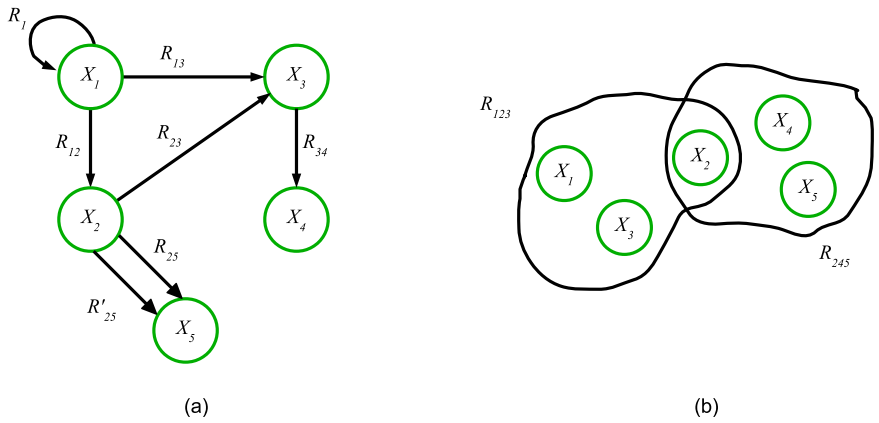


Figura 2.4: Ejemplos de: grafo para CSP binario (a) e hipergrafo para CSP no-binario (b).

Ejemplo. En la Figura 2.4 (a), muestra un grafo de restricciones que representa a un CSP con cinco variables: X_1, X_2, X_3, X_4, X_5 y siete restricciones: donde R_1 es una restricción unaria, las restricciones R_{12}, R_{13}, R_{23} y R_{34} son restricciones binarias y R_{25}, R'_{25} son restricciones binarias no-normalizadas. R_{25} y R'_{25} hacen que el CSP sea no-normalizado. El sentido de la flecha en el grafo indica como fue expresada la restricción en el problema original. La Figura 2.4 (b), muestra un

ejemplo de hipergrafo, para un CSP no-binario de dos restricciones: R_{123} y R_{245}, la primera compuesta por las variables $\{X_1, X_2, X_3\}$, y la segunda por las variables $\{X_2, X_4, X_5\}$.

Definición 2.2.15. El *grado de una variable* X_i en un CSP es el número de restricciones en las que X_i participa. La notación que se empleará para indicar el grado de la variable X_i es: $grado(X_i) = a$, donde $a \in \mathbb{N}$.

Así, para el ejemplo de la Figura 2.4, los grados de las variables son: $grado(X_1) = 3$; $grado(X_2) = 4$; $grado(X_3) = 3$; $grado(X_4) = 1$ y $grado(X_5) = 2$.

2.2.4. Árbol de Búsqueda

El conjunto formado por todas las posibles asignaciones a un CSP, se denomina *espacio de estados* y puede ser representado en un árbol, que se denomina árbol de búsqueda. Los algoritmos de resolución de CSPs recorren de diferentes formas el árbol de búsqueda. Por ejemplo, la búsqueda mediante backtracking[1] recorre el árbol mediante *exploración primero en profundidad*. Si se asume que el orden de las variables es estático y que dicho orden no cambia durante el proceso de búsqueda, entonces un nodo en el nivel k en el árbol de búsqueda representa un estado donde las variables $X_1, ..., X_k$ están instanciadas (asignadas) y el resto $X_{k+1}, ..., X_n$ no lo están. De esta forma, se puede asignar cada nodo en el árbol de búsqueda con la tupla consistente de todas las instanciaciones llevadas a cabo.

Así, la raíz del árbol de búsqueda representa la tupla vacía, donde ninguna variable tiene asignado valor alguno. Los nodos en el primer nivel son *1-tuplas* que representan estados donde se le 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 dicha *n-tupla* es una solución del problema.

[1] El algoritmo de backtracking es explicado con más detalle en una sección posterior.

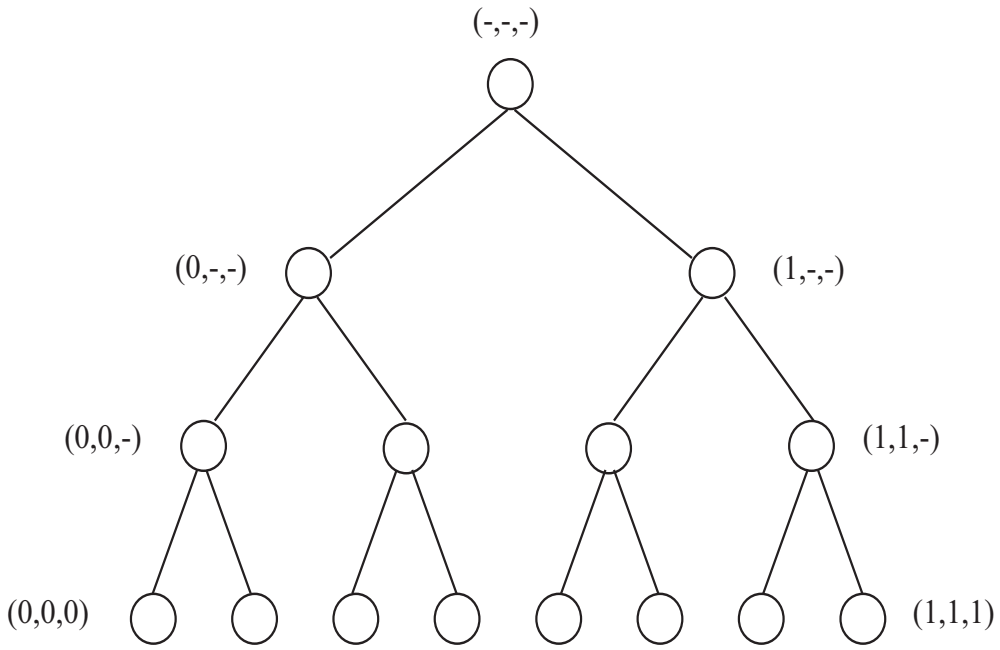


Figura 2.5: Árbol de búsqueda en un CSP binario.

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. El padre de un nodo $(a_1, ..., a_i)$ es el nodo que consiste en todas las asignaciones hechas en el nodo $(a_1, ..., a_i)$ excepto la última, es decir, el padre del nodo $(a_1, ..., a_i)$ es el nodo $(a_1, ..., a_{i-1})$. Un antecesor de un nodo $(a_1, ..., a_i)$ es un nodo $(a_1, ..., a_j)$, donde $j < i$. Así, el nodo $(a_1, ..., a_i)$ se llamará descendiente del nodo $(a_1, ..., a_j)$. Un hijo del nodo $(a_1, ..., a_i)$ es un nodo con el mismo conjunto de asignaciones que el nodo $(a_1, ..., a_i)$ mas una asignación de la siguiente variable, es decir, un nodo de la forma $(a_1, ..., a_{i+1})$. Un nodo hoja en un problema con n variables es cualquier nodo $(a_1, ..., a_n)$. Los nodos que se encuentran próximos a la raíz, se llaman nodos superficiales. Los nodos próximos a las hojas del árbol de búsqueda se llaman nodos profundos.

En la Figura 2.5, presentamos un ejemplo de árbol de búsqueda de un CSP compuesto con tres variables, cuyos dominios son 0 y 1 [22]. Cada nivel en el árbol corresponde a una variable. 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, un nodo hoja, que se encuentra a la derecha en el árbol de búsqueda es $(1, 1, 1)$, donde todas las variables toman el valor 1. El nodo $(1, 1, -)$ es el nodo padre de $(1, 1, 1)$, lo que indica que el nodo $(1, 1, 1)$ es el hijo del nodo $(1, 1, -)$. El nodo $(1, -, -)$ es un antecesor del nodo $(1, 1, 1)$, por lo que $(1, 1, 1)$ es un descendiente del nodo $(1, -, -)$.

En la búsqueda primero en profundidad del árbol de búsqueda, la variable correspondiente al nivel actual se denomina *variable actual*. Las variables correspondientes a niveles menos profundos se denominan *variables pasadas*. Las variables restantes que aún no se han instanciado se denominan *variables futuras*.

2.3. Técnicas de Consistencia

Las técnicas de consistencia se introdujeron por primera vez en IA para mejorar la eficiencia de los programas de reconocimiento de imágenes [123]. Como hemos comentado en el Capítulo 1, en la literatura se han propuesto diversas técnicas de *consistencia local* como formas para mejorar la eficiencia de los algoritmos de búsqueda. Estas técnicas se usan como etapas de pre-proceso donde se detectan y se eliminan las inconsistencias locales, antes de empezar la búsqueda o durante el proceso de búsqueda en sí, con el fin de reducir los nodos a instanciar en el árbol de búsqueda.

La Figura 2.6 amplía el conjunto de técnicas de inferencia mostrado en la Figura 2.1 y muestra las diferentes técnicas y algoritmos asociados a cada una de ellas. En dicha figura podemos diferenciar diversos niveles de consistencia local como nodo-consistencia, arco-consistencia o path-consistencia. Los algoritmos que alcanzan tales niveles de consistencia eliminan instanciaciones de valores en los dominios de las variables que son inconsistentes, es decir descartan nodos del árbol de búsqueda, que no pueden participar en ninguna solución.

A tales algoritmos se les llama *algoritmos de propagación de restricciones*, *algoritmos de filtrado* o *algoritmos de consistencia*. Aunque aplicando los algoritmos de consistencia, no garantizamos que todos los pares variable-valor restantes formen parte de una solución, la práctica ha demostrado que pueden ser muy útiles como una etapa de pre-proceso, para reducir la complejidad de los CSPs y además durante la búsqueda, para podar el espacio de búsqueda.

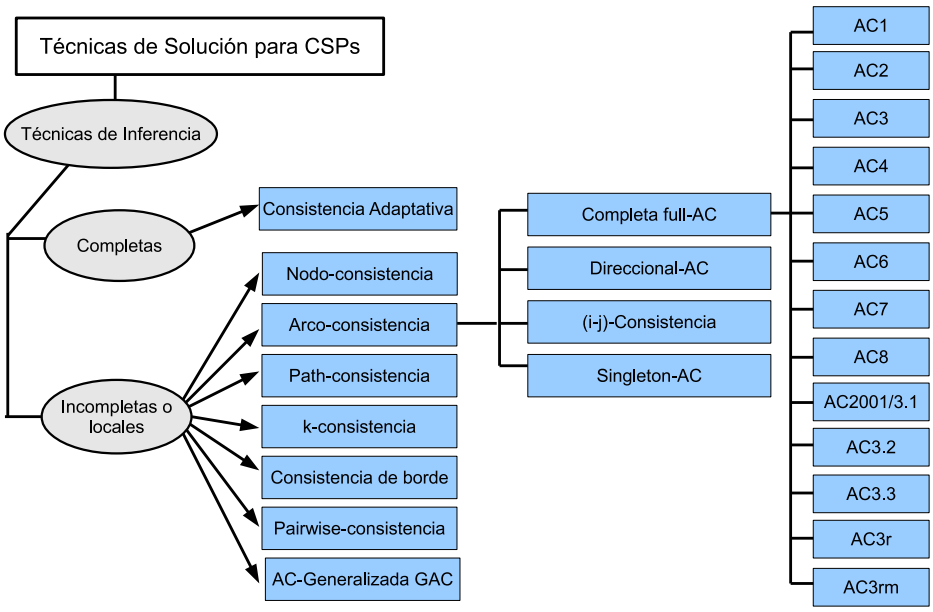


Figura 2.6: Clasificación de las técnicas de inferencia completas y locales.

En esta sección, revisaremos algunos trabajos previos sobre algoritmos de consistencia.

2.3.1. Nodo-Consistencia

El nivel básico de consistencia es la nodo-consistencia, donde se eliminan los valores inconsistentes con las restricciones unarias donde participa la variable.

Definición 2.3.1. Una *variable es nodo-consistente* si y solo si todos los valores que están en su dominio satisfacen la restricción unaria donde participa la variable: $\forall X_i \in X, \forall R_i \in R, \exists a \in D_i : a$ satisface R_i.

Definición 2.3.2. Un *CSP es nodo-consistente* sii todas sus variables son nodo-consistentes: $\forall X_i \in X, \forall R_i \in R : X_i$ satisface R_i. Un CSP es nodo-inconsistente si una (o más) de sus variables no es nodo-consistente.

Ejemplo. Sea un problema de una variable X_1, cuyo dominio discreto es: $D_1 = \{-1, 0, 2, 4, 5, 6\}$ y la restricción unaria $R_1 : X_1 > 2$. La consistencia de nodo eliminará los valores $\{-1, 0, 2\}$ del dominio de X_1. En la Figura 2.7 (a) se muestra el problema original y en la Figura 2.7 (b) se muestra el resultado de aplicar nodo-consistencia a la variable X_1.

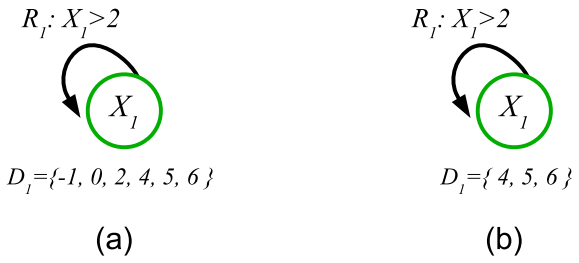


Figura 2.7: Nodo-consistencia: (a) dominio original y (b) dominio nodo-consistente.

El Algoritmo 1, muestra un algoritmo para realizar la nodo-consistencia. Consta de un ciclo que recorre cada una de las restricciones unarias y poda del dominio de la variable los valores inconsistentes con la restricción unaria. Muchos de los algoritmos que realizan arco-consistencia, asumen que los dominios son nodo-consistentes o, dentro del algoritmo, indican la realización del proceso de nodo-consistencia, antes de efectuar el proceso de arco-consistencia.

Algoritmo 1: Procedimiento Nodo-consistencia

Datos: CSP, $P = \langle X, D, R \rangle$
Resultado: **verdadero** y P' (el cual es nodo-consistente) o **falso** y P' (el cual es nodo-inconsistente porque algún dominio se ha quedado vacío).

```
1 principio
2     para cada arco R_i ∈ R hacer
3         para cada valor a ∈ D_i hacer
4             si ⟨X_i, a⟩ no satisface R_i entonces
5                 Eliminar a de D_i
6         si D_i = ∅ entonces
7             devolver falso
8     devolver verdadero
9 fin
```

2.3.2. Arco-Consistencia

La arco-consistencia es una técnica de inferencia incompleta que se aplica a restricciones compuestas por dos variables. La mayoría del trabajo realizado sobre problemas de propagación de restricciones se han centrado en la arco-consistencia, porque alcanzar la arco-consistencia (o consistencia de arco) es una forma 'económica' de eliminar algunos valores que no forman parte de la solución. Los algoritmos de arco-consistencia están basados en la noción de soporte. Estos algoritmos aseguran que cada valor en el dominio de cada variable está soportado por un valor de la otra variable que participa en la restricción. Se ha demostrado que los algoritmos de arco-consistencia son muy efectivos cuando se aplican como técnica de pre-proceso y también durante el proceso de búsqueda.

Definición 2.3.3. Un *valor* $a \in D_i$ es arco-consistente relativo a X_j sii existe un valor $b \in D_j$ tal que $\langle X_i, a\rangle$ y $\langle X_j, b\rangle$ satisfacen la restricción R_{ij}.

Definición 2.3.4. Una *variable* X_i es arco-consistente relativa a X_j sii todos los valores en D_i son arco-consistentes relativos a X_j, i.e., una variable es arco-consistente si es consistente relativa a todas aquellas con las que interviene en alguna restricción.

Definición 2.3.5. Un *CSP* es arco-consistente sii todas las variables son arco-consistentes, e.g., todos los arcos R_{ij} y R'_{ji} son arco-consistentes. Se debe tener en cuenta que en esta definición estamos hablando de arco-consistencia completa. Un CSP es arco-inconsistente si una (o más) de sus variables no es arco-consistente.

Proponer algoritmos eficientes que realicen la arco-consistencia ha sido siempre considerado como un punto central en la comunidad de satisfacción de restricciones [28]. Así, partiendo de la idea de Fikes, durante más de dos décadas se han propuesto una variedad de algoritmos que establecen la arco-consistencia [77] tales como: AC1, AC2 y AC3 [83]; AC4 [89]; AC5 [93]; AC6 [29]; AC7 [30]; AC8 [39]; AC2001/3.1 [33]; AC3.2 y AC3.3 [76] AC3rm [77]; AC3^{bit} [79]; L, L', L'' [120]; MAC, MAC2001, MACrm [77, 80], etc. Sin embargo AC3 y AC4 son los más utilizados [24] y en los últimos años la investigación se ha centrado en refinar los algoritmos AC3 y AC2001/3.1. Cabe destacar que la mayoría de los algoritmos desarrollados asumen

que las restricciones son binarias, normalizadas y asumen que el problema es nodo-consistente.

Se le llama *granularidad* a la forma en la que los algoritmos de arco-consistencia realizan las propagaciones [28]. Las propagaciones consisten en volver a revisar arcos o valores que se hayan podido volver inconsistentes a causa de la poda de uno o más de sus soportes. Así, los algoritmos: AC1, AC2, AC3; AC2001/3.1, AC3.2, AC3.3 y AC3rm realizan sus propagaciones a nivel de restricciones y por ello, son llamados algoritmos de grano-grueso (del inglés coarse-grained algorithm), mientras que los algoritmos: AC4, AC6, AC7 y AC8 realizan sus propagaciones a nivel de valores y son llamados algoritmos de grano-fino (del inglés fine-grained algorithm). El algoritmo AC5 por ser implementado sobre AC3 y AC4 hará las propagaciones en función de dichos algoritmos.

La Figura 2.8 muestra los algoritmos presentados cronológicamente, donde por ejemplo, AC3→AC2 indica que AC3 está basado en AC2. Adicionalmente, la Figura 2.8 agrupa a los algoritmos en tres categorías: grano-grueso, grano-fino e híbrido de acuerdo a la técnica de propagación utilizada, así por ejemplo, AC3 está colocado dentro de la banda de grano-grueso ya que realiza la propagación a nivel de restricciones.

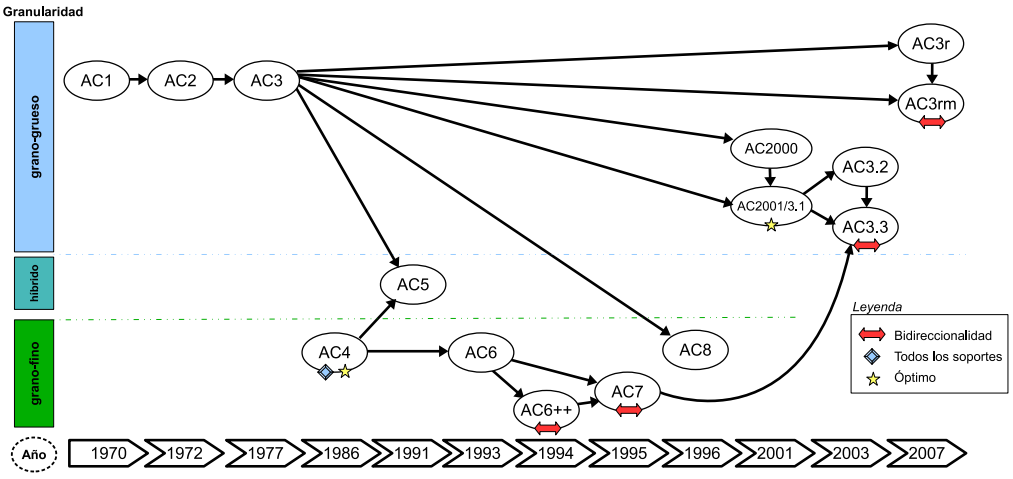


Figura 2.8: Evolución y clasificación de los algoritmos de arco-consistencia.

Los algoritmos de arco-consistencia se han enfocado en disminuir las complejidades espacial y temporal. La Tabla 2.1, muestra la complejidad espacial y temporal de los algoritmos de arco-consistencia presentes en la literatura.

Tabla 2.1: Complejidad espacial y temporal de los algoritmos de arco-consistencia.

Algoritmo	*Complejidad espacial*	*Complejidad Temporal*
AC1	$O(n^3d^3)$	$O(ned^3)$
AC2	$O(e)$	$O(ed^3)$
AC3	$O(e)$	$O(ed^3)$
AC4	$O(ed^2)$	$O(ed^2)$
AC5*	$O(ed)$	$O(ed)$
AC6	$O(ed)$	$O(ed^2)$
AC6++	$O(ed)$	$O(ed^2)$
AC7	$O(ed^2)$	$O(ed^2)$
AC8	$O(n)$	$O(ed^3)$
AC2000	$O(ed)$	$O(ed^3)$
AC2001/3.1	$O(ed)$	$O(ed^2)$
AC3.2	$O(ed)$	$O(ed^2)$
AC3.3	$O(ed)$	$O(ed^2)$
AC3r	$O(ed)$	$O(ed^3)$
AC3rm	$O(ed)$	$O(ed^3)$
Key: e = arcos; d = talla del dominio más grande; n = variables; * para restricciones funcionales y monotónicas		

Para demostrar su eficiencia, los algoritmos propuestos se comparan con los algoritmos de consistencia previamente desarrollados. La Tabla 2.2 muestra el tipo de evaluaciones efectuadas en los algoritmos de arco-consistencia (al momento de su publicación). Las evaluaciones teóricas se limitan a medir la complejidad espacial y temporal. Las evaluaciones empíricas, además de la complejidad, miden el rendimiento sobre el tiempo de ejecución (CPU) y el número de chequeos de restricciones.

Las principales mejoras en los algoritmos de arco-consistencia se han alcanzado:

- cambiando la forma de propagación: de arcos (restricciones) a valores (es decir, cambiando la granularidad, de grano-grueso a grano-fino);
- añadiendo nuevas estructuras: que les permitan almacenar información posteriormente útil;
- realizando el chequeo de soportes bidireccionalmente (como los algoritmos AC6++, AC7, AC3.3);

- cambiando el número de soportes buscados: buscando todos los soportes (AC4) o buscando sólo los soportes necesarios (AC3, AC6, AC7, AC8 y AC2001 limitan la búsqueda de soportes a 1);
- mejorando el proceso de chequeos de restricciones (es decir, realizar el chequeo de restricciones únicamente cuando sea necesario (AC7 y AC2001/3.1);
- cambiando el soporte que se almacena: ninguno (AC3), el primer soporte encontrado (AC2001/3.1) o el soporte residual encontrado (AC3r y AC3rm), etc.

Tabla 2.2: Tipos de evaluación realizada en los algoritmos de arco-consistencia. En la evaluación empírica -adicional a la complejidad temporal y espacial- es medido: el tiempo de cómputo y el número de chequeos de restricciones.

Algoritmo	AC1	AC2	AC3	AC4	AC5	AC6	AC6++	AC7	AC8	AC2000	AC2001/3.1	AC3.2	AC3.3	AC3r	AC3rm
AC2	T														
AC3	T	T													
AC4			T												
AC5*			T	T											
AC6			E	E											
AC6++			E	E		E									
AC7			E	E		E	E								
AC8			E	E		E									
AC2000			E			E					E				
AC2001/3.1			E			E				E					
AC3.2			E								E		E		
AC3.3			E								E	E			
AC3r			E								E				E
AC3rm			E								E	E		E	

Lave: E = evaluación empírica; T = evaluación teórica

A continuación, describiremos en detalle los algoritmos de arco-consistencia, sobre los cuáles se ha basado esta investigación. Todos suponen que el CSP es nodo-consistente, binario y normalizado.

AC3

El algoritmo AC3 [83] propuesto por Mackworth, es uno de los algoritmos de arco-consistencia más conocido y utilizado, debido a su simplicidad y a la ligereza de estructuras (sólo requiere de una cola Q, donde almacena las restricciones pendientes de evaluar). AC3 pertenece a la categoría de grano-grueso, ya que realiza sus propagaciones a nivel de restricciones. El algoritmo AC3 es el marco de algoritmos de grano-grueso, que han aparecido posteriormente, tales como: AC2001/3.1, AC3.2, AC3.3, ACr y ACrm. Así mismo, AC3 es referencia en las comparaciones de eficiencia de los algoritmos que le suceden (ver Tabla 2.2).

El cuerpo principal de AC3 (ver Algoritmo 2) consta de un bucle que selecciona y revisa (utilizando el Algoritmo 3) las restricciones almacenadas en una cola Q hasta que no ocurran cambios (Q está vacío) o se torne vacío el dominio de una variable. El primer caso asegura que todos los valores de los dominios son consistentes con todas las restricciones. En el segundo caso indica que el problema no tiene solución.

Algoritmo 2: Procedimiento AC3

```
   Datos: CSP, P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es arco-consistente) o falso y P' (el cual es
              arco-inconsistente porque algún dominio se ha quedado vacío)
 1 principio
 2     para cada arco R_ij ∈ R hacer
 3         Añadir (Q, R_ij)
 4         Añadir (Q, R'_ji)
 5     mientras Q ≠ φ hacer
 6         Seleccione y borre R_ij de la cola Q
 7         si ReviseAC3(R_ij) = verdadero entonces
 8             si D_i ≠ φ entonces
 9                 Añadir (Q, R_ki) con k ≠ i, k ≠ j
10             sino
11                 devolver falso /*dominio vacio*/
12     devolver verdadero
13 fin
```

Para evitar llamadas innecesarias al procedimiento *Revise*, AC3 almacena en

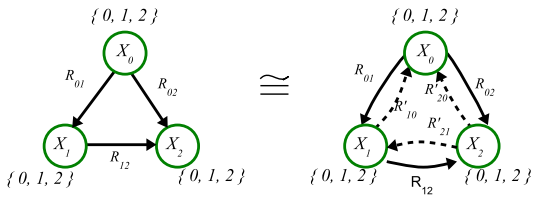
Algoritmo 3: Procedimiento ReviseAC3

Datos: CSP P' definido por dos variables $X = \{X_i, X_j\}$, dominios $D = \{D_i, D_j\}$, y una restricción $R = \{R_{ij}\}$.
Resultado: D_i, tal que X_i es arco-consistente relativo a X_j y la variable booleana *change* donde *change* = **verdadero** indica que el dominio D_i ha sido podado y *change* =**falso** indica que el dominio D_i no ha sido modificado.

```
1 principio
2     change ← falso
3     para cada a ∈ D_i hacer
4         si ∄b ∈ D_j tal que (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ R_ij entonces
5             Eliminar a de D_i
6             change ← verdadero
7     devolver change
8 fin
```

la cola Q todas las restricciones R_{ij} que no tienen garantía de que D_i sea arco-consistente. Además, Q es actualizada añadiendo las restricciones R_{ki}, las cuales fueron previamente evaluadas, donde los valores de D_k pueden ser inconsistentes porque D_i fue podado. El algoritmo AC3 tiene una complejidad temporal de $O(ed^3)$ y una complejidad espacial de $O(e)$ [84], donde d es la talla del dominio y e es el número de restricciones en el problema.

La complejidad temporal de AC3 no es óptima porque el procedimiento *Revise* (ver Algoritmo 3) no recuerda nada de sus cómputos al encontrar los soportes de los valores, lo cual hace que AC3 repita los mismos chequeos de restricciones varias veces.



Valores iniciales de R		
Restricción		
R_{ij}	X_i *op* X_j	dirección
R_{01}	$X_0 < X_1$	directa
R'_{10}	$X_1 > X_0$	inversa
R_{02}	$X_0 < X_2$	directa
R'_{20}	$X_2 > X_0$	inversa
R_{12}	$X_1 < X_2$	directa
R'_{21}	$X_2 > X_1$	inversa

Figura 2.9: Ejemplo de CSP binario normalizado con tres variables.

Para el ejemplo de la Figura 2.9, la cola Q inicialmente almacena las restricciones, intercalando restricciones directas e inversas, como se muestra en la Tabla de la Figura 2.9 (derecha). Para alcanzar la arco-consistencia de este ejemplo, AC3

requirió realizar 9 iteraciones, ya que AC3 evalúa primeramente las seis restricciones que fueron almacenadas inicialmente en Q (Ver Tabla 2.3, iteraciones 1 - 6). Las restricciones que requieren ser re-evaluadas (R_{01}, R_{02}, y R_{20}) fueron añadidas al final de la cola Q (Véase Tabla 2.3, iteraciones 5 - 7). Estas restricciones fueron evaluadas en las iteraciones 7, 8, y 9, respectivamente.

Tabla 2.3: Iteraciones realizadas por AC3 almacenando las propagaciones en una cola para el ejemplo de Figura 2.9.

Iteración	R_i	Restricción X_i *op* X_j	Arco (X_i, X_j)	Poda X_i	Restricción Propagación
1	R_{01}	$X_0 < X_1$	(X_0, X_1)	$X_0 = 2$	
2	R_{10}	$X_1 > X_0$	(X_1, X_0)	$X_1 = 0$	
3	R_{02}	$X_0 < X_2$	(X_0, X_2)	*Ninguna*	
4	R_{20}	$X_2 > X_0$	(X_2, X_0)	$X_2 = 0$	
5	R_{12}	$X_1 < X_2$	(X_1, X_2)	$X_1 = 2$	R_{01}
6	R_{21}	$X_2 > X_1$	(X_2, X_1)	$X_2 = 1$	R_{02}
7	R_{01}	$X_0 < X_1$	(X_0, X_1)	$X_0 = 1$	R_{20}
8	R_{02}	$X_0 < X_2$	(X_0, X_2)	*Ninguna*	
9	R_{20}	$X_2 > X_0$	(X_2, X_0)	*Ninguna*	

La propagación de la iteración 6 (R_{21}) fue ineficiente porque no efectuó ninguna poda (véase Tabla 2.3, iteracion 8). Similarmente, en la iteración 7, la restricción R_{20} fue añadida a la cola Q. R_{20} fue evaluada en la iteración 9, pero su evaluación no realizó ninguna poda.

Por otro lado, se modificó el comportamiento de Q de cola a pila. Esto fue llevado a cabo para comparar el rendimiento de AC3, según el modo de almacenamiento de las propagaciones. Así, la Tabla 2.4 muestra el comportamiento de AC3 para el mismo ejemplo de la Figura 2.9, pero Q es manejada como una pila. Las restricciones que deben ser re-evaluadas (R_{01}, R_{20} y R_{21}) fueron añadidas al tope de Q (vea Tabla 2.4, iteraciones 5, 6 y 8), por lo que esas restricciones fueron evaluadas en la siguiente iteración ya que se simuló una pila. No obstante, hay propagaciones de restricciones innecesarias. Las iteraciones inefectivas son la 7 y la 9 de la Tabla 2.4. La evaluación empírica entre AC3 (cola) y AC3 (pila) en problemas más complejos indicaron que AC3 (cola) tuvo un mejor rendimiento.

AC4

El algoritmo AC4 [89] es conjuntamente con AC3 uno de los algoritmos de arco-consistencia más utilizado (ver Algoritmo 4). AC4 realiza una búsqueda exhaustiva

Tabla 2.4: Iteraciones llevadas a cabo por AC-3 almacenando en pila para el ejemplo de la Figura 2.9.

Iter	R_{ij}	Restriccion X_i op X_j	Arco (X_i, X_j)	Podas X_i	Add Restricción.
1	R_{01}	$X_0 < X_1$	(X_0, X_1)	$X_0 = 2$	
2	R_{10}	$X_1 > X_0$	(X_1, X_0)	$X_1 = 0$	
3	R_{02}	$X_0 < X_2$	(X_0, X_2)	*Ninguna*	
4	R_{20}	$X_2 > X_0$	(X_2, X_0)	$X_2 = 0$	
5	R_{12}	$X_1 < X_2$	(X_1, X_2)	$X_1 = 2$	R_{01}
6	R_{01}	$X_0 < X_1$	(X_0, X_1)	$X_0 = 1$	R_{20}
7	R_{20}	$X_2 > X_0$	(X_2, X_0)	*Ninguna*	
8	R_{21}	$X_2 > X_1$	(X_2, X_1)	$X_2 = 1$	R_{02}
9	R_{02}	$X_0 < X_2$	(X_0, X_2)	*Ninguna*	

de soportes, ya que busca todos los soportes que tiene cada uno de los valores de los dominios de las variables. AC4 fue propuesto para mejorar la complejidad temporal de AC3 [89] y es el único algoritmo que confirma la existencia de un soporte sin identificarlo a través de un proceso búsqueda. De esta manera, AC4 es el primer algoritmo de la categoría *grano-fino (fine-grained algorithms)* porque las propagaciones las realiza a nivel de valores [28]. AC4 guarda la mayor cantidad de información en la etapa de pre-proceso, para evitar rehacer los mismos chequeos de restricciones durante la propagación de la poda de valores. Además, AC4 tiene una complejidad temporal óptima de $O(ed^2)$ y una complejidad espacial de $O(ed^2)$, donde e es el número de restricciones y d es la talla máxima del dominio. AC4 ha influenciado directamente a los algoritmos de arco-consistencia AC5, AC6 y AC7.

Con el fin de realizar el chequeo de restricciones una sola vez y para identificar los valores relevantes que requieren ser re-examinados, AC4 requiere de las siguientes estructuras de datos:

- S es una matriz $S[X_j, b]$ que contiene la lista de instanciaciones $\langle X_i, a\rangle$ soportadas por $\langle X_j, b\rangle$.
- $Counter$ es una matriz $Counter[X_i, a, X_j]$ que contiene el número de soportes para el valor $a \in D_i$ en la variable X_j.
- M es una matriz $M[X_i, a]$ que contiene el valor 1 si el valor $a \in D_i$ o contiene el valor 0 si el valor $a \notin D_i$ (indicando que la instanciación $\langle X_i, a\rangle$ ha sido podada de D_i).
- Q es una cola que almacena tuplas $\langle X_i, a\rangle$ (instanciaciones eliminadas) que

Algoritmo 4: Procedimiento AC4

Datos: CSP $P = \langle X, D, R\rangle$
Resultado: **verdadero** y P' (el cual es arco-consistente) o **falso** y P' (el cual es arco-inconsistente)

```
1  principio
2      InitilizeAC4(P)
3      si initial = verdadero entonces
4          mientras Q ≠ φ hacer
5              Seleccione y elimine ⟨X_j, b⟩ de la cola Q
6              para cada ⟨X_i, a⟩ ∈ S[X_j, b] hacer
7                  Counter[X_i, a, X_j] ← Counter[X_i, a, X_j] − 1
8                  si Counter[X_i, a, X_j] = 0 ∧ M[X_i, a] = 1 entonces
9                      Elimine a de D_i
10                     Q ← Q ∪ ⟨X_i, a⟩
11                     M[X_i, a] ← 0
12                 si D_i = φ entonces
13                     devolver falso
14         devolver verdadero
15     sino
16         devolver falso
17 fin
```

Algoritmo 5: Procedimiento InitializeAC4

Datos: CSP $P = \langle X, D, R\rangle$ /*R contiene las restricciones directas e inversas*/
Resultado: Variable booleana *initial* (donde *initial* =**verdadero** indica que ningún dominio se ha quedado vacío e *initial* = **falso** indica que P' es arco-inconsistente); cola Q; matrices S, M, *Counter* y CSP P' actualizados.

```
1  principio
2      Q ← {}
3      S[X_j, b] ← {}   /* ∀X_j ∈ X ∧ ∀b ∈ D_j */
4      M[X_i, a] ← 1   /* ∀X_i ∈ X ∧ ∀a ∈ D_i */
5      Counter[X_i, a, X_j] ← 0   /* ∀X_i, X_j ∈ X, i ≠ j ∧ ∀a ∈ D_i */
6      para cada arco R_ij ∈ R hacer
7          para cada valor a ∈ D_i hacer
8              total ← 0
9              para cada valor b ∈ D_j hacer
10                 si (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ R_ij entonces
11                     total ← total + 1
12                     Añadir (S[X_j, b], ⟨X_i, a⟩)
13             si total = 0 entonces
14                 Eliminar a de D_i
15                 Q ← Q ∪ ⟨X_i, a⟩
16                 M[X_i, a] ← 0
17             sino
18                 Counter[X_i, a, X_j] ← total
19             si D_i = φ entonces
20                 devolver initial ← falso
21     devolver initial ← verdadero
22 fin
```

requieren un procesamiento posterior.

El cuerpo principal del algoritmo AC4 (ver Algoritmo 4) consta de dos fases: Inicialización de las estructuras de datos (ver paso 2, donde se invoca al Algoritmo 5) y Propagación (pasos 3 - 14). El Algoritmo 5 tiene la finalidad de inicializar las estructuras $S, Counters, M$ y Q. Estas inicializaciones son utilizadas para: recordar parejas de valores consistentes de las variables (matrix S); contar valores 'soportados' del dominio de la variable (matriz $Counter$); eliminar aquellos valores que no tienen ningún soporte y recordar la remoción (matriz M y cola Q, respectivamente). Una vez que el Algoritmo 5 ha finalizado, el Algoritmo 4 comienza la fase de propagación. Esta fase consiste en propagar las consecuencias de los valores podados en el resto de las variables. Así, las tuplas ⟨variable, valor⟩ almacenadas en Q son seleccionadas y procesadas hasta que: no ocurran más cambios (Q está vacía), o se vacíe el dominio de una variable. En el primer caso, el algoritmo asegura que todos los valores de los dominios son consistentes con todas las restricciones (son arco-consistentes), mientras que en el segundo caso, el algoritmo indica que el problema no tiene solución.

Para el ejemplo mostrado en la Figura 2.9, R almacena las restricciones según se muestra en la Tabla de la Figura 2.9 (derecha). La Tabla 2.5 muestra como el procedimiento *InitializeAC4* evalúa cada restricción. Las tuplas de valores que deben ser re-evaluadas ($\langle X_0, 2\rangle, \langle X_2, 0\rangle, \langle X_1, 0\rangle, \langle X_1, 2\rangle$, y $\langle X_2, 1\rangle$) fueron añadidas a la cola Q (Ver Tabla 2.5, iteraciones 1,2,4,5 y 6). Los valores de M, $Counter$ y S (posteriores a la fase de Inicialización) son mostrados en las columnas **Ini** de las Tablas 2.6, 2.7 y 2.8.

Posteriormente a la fase de inicialización, se lleva a cabo la propagación con las tuplas almacenadas en Q. La Tabla 2.9 muestra ese proceso. Puede observarse que las iteraciones 1, 2 y 3 realizan propagaciones ineficientes porque las tuplas $\langle X_0, 2\rangle$, $\langle X_2, 0\rangle$, $\langle X_1, 0\rangle$ no soportan ningún valor de otra variable (Ver S en la Tabla 2.8, en la columna Ini, las filas vacías).

Los valores de M, S y $Counter$ (posteriores a la fase de propagación) se muestran en la columna **Prop** de las Tablas 2.6 y 2.7.

En el ejemplo mostrado, AC4 lleva a cabo 33 chequeos de restricciones (Cc) con el Algoritmo 5. Lleva a cabo 6 propagaciones (Np) en Q con el Algoritmo 4 y realiza 6 podas de valores para alcanzar la arco-consistencia.

Tabla 2.5: Iteraciones realizadas por el procedimiento *InitializeAC4* para el ejemplo mostrado en la Figura 2.9.

Iter	Restricción $R_{ij} : X_i$ op X_j	val a	val b	var total	Poda X_i	Add Q
1	$R_{02} : X_0 < X_2$	0	0	0		
			1	1		
			2	2		
		1	0	0		
			1	0		
			2	1		
		2	0	0		
			1	0		
			2	0	$X_0 = 2$	$\langle X_0, 2\rangle$
2	$R'_{20} : X_2 > X_0$	0	0	0		
			1	0	$X_2 = 0$	$\langle X_2, 0\rangle$
		1	0	1		
			1	1		
		2	0	1		
			1	2		
3	$R_{01} : X_0 < X_1$	0	0	0		
			1	1		
			2	2		
		1	0	0		
			1	0		
			2	1		
4	$R'_{10} : X_1 > X_0$	0	0	0		
			1	0	$X_1 = 0$	$\langle X_1, 0\rangle$
		1	0	1		
			1	1		
		2	0	1		
			1	2		
5	$R_{12} : X_1 < X_2$	1	1	0		
			2	1		
		2	1	0		
			2	0	$X_1 = 2$	$\langle X_1, 2\rangle$
6	$R'_{21} : X_2 > X_1$	1	1	0	$X_2 = 1$	$\langle X_2, 1\rangle$
		2	1	1		

Tabla 2.6: Cambios en la matriz M después de las fases de Inicialización (Ini) y de Propagación (Prop) realizadas por el procedimiento *AC4* para el ejemplo de la Figura 2.9.

$M[var, val]$	**Start**	Ini	Prop
$M[X_0, 0]$	1		
$M[X_0, 1]$	1		0
$M[X_0, 2]$	1	0	
$M[X_1, 0]$	1	0	
$M[X_1, 1]$	1		
$M[X_1, 2]$	1	0	
$M[X_2, 0]$	1	0	
$M[X_2, 1]$	1	0	
$M[X_2, 2]$	1		

Tabla 2.7: Cambios en la matriz *Counter* después de las fases de Inicialización (Ini) y de Propagación (Prop) realizadas por el procedimiento *AC4* para el ejemplo de la Figura 2.9.

$Counter[X_i, a, X_j]$	**Comienzo**	Ini	Prop
$Counter[X_0, 0, X_2]$	0	2	1
$Counter[X_0, 1, X_2]$	0	1	
$Counter[X_0, 2, X_2]$	0	0	
$Counter[X_0, 0, X_1]$	0	2	1
$Counter[X_0, 1, X_1]$	0	1	0
$Counter[X_0, 2, X_1]$	0		
$Counter[X_1, 0, X_0]$	0	0	
$Counter[X_1, 1, X_0]$	0	1	
$Counter[X_1, 2, X_0]$	0	2	1
$Counter[X_1, 0, X_2]$	0		
$Counter[X_1, 1, X_2]$	0	1	
$Counter[X_1, 2, X_2]$	0	0	
$Counter[X_2, 0, X_0]$	0	0	
$Counter[X_2, 1, X_0]$	0	1	
$Counter[X_2, 2, X_0]$	0	2	1
$Counter[X_2, 0, X_1]$	0		
$Counter[X_2, 1, X_1]$	0	0	
$Counter[X_2, 2, X_1]$	0	1	

Tabla 2.8: Cambios en la matriz S después de las fases de Inicialización (Ini) y de Propagación (Prop) realizadas por el procedimiento *AC4* para el ejemplo de la Figura 2.9.

$S[X_j, b]$	**Comienzo**	Ini
$S[X_0, 0]$	{}	$\{\langle X_2, 1\rangle, \langle X_2, 2\rangle, \langle X_1, 1\rangle, \langle X_1, 2\rangle\}$
$S[X_0, 1]$	{}	$\{\langle X_2, 2\rangle, \langle X_1, 2\rangle\}$
$S[X_0, 2]$	{}	
$S[X_1, 0]$	{}	
$S[X_1, 1]$	{}	$\{\langle X_0, 0\rangle, \langle X_2, 2\rangle\}$
$S[X_1, 2]$	{}	$\{\langle X_0, 0\rangle, \langle X_0, 1\rangle\}$
$S[X_2, 0]$	{}	
$S[X_2, 1]$	{}	$\{\langle X_0, 0\rangle\}$
$S[X_2, 2]$	{}	$\{\langle X_0, 0\rangle, \langle X_0, 1\rangle, \langle X_1, 1\rangle\}$

Tabla 2.9: Iteraciones realizadas por AC4 (sólo propagación) para el ejemplo de la Figura 2.9.

Iter	tupla $\langle X_j, b\rangle$	Soportes $S[X_j, b]$	Counter $[X_i, a, X_j]$	Podas X_i	Add Q $\langle X_i, a\rangle$
1	$\langle X_0, 2\rangle$	{}			
2	$\langle X_2, 0\rangle$	{}			
3	$\langle X_1, 0\rangle$	{}			
4	$\langle X_1, 2\rangle$	$\langle X_0, 0\rangle$	1		
		$\langle X_0, 1\rangle$	0	$\langle X_0, 1\rangle$	$\langle X_0, 1\rangle$
5	$\langle X_2, 1\rangle$	$\langle X_0, 0\rangle$	1		
6	$\langle X_0, 1\rangle$	$\langle X_2, 2\rangle$	1		
		$\langle X_1, 2\rangle$	1		

A pesar de lo completas que son las estructuras de datos que mantiene AC4 durante la búsqueda, hemos detectado ineficiencias cuando son actualizadas:

- No es necesario chequear la restricción inversa cuando la restricción directa puede almacenar toda la información (corregida por AC7, pero limitando la búsqueda a un único soporte).
- Hay propagaciones de tuplas que son inefectivas porque dichas tuplas no soportan ningún otro valor de otra variable (no corregida por ningún algoritmo de arco-consistencia posterior).

AC6

El algoritmo AC6, propuesto por Bessiere y Cordier en [29], es un algoritmo de grano-fino. AC6 (ver Algoritmo 6) está basado en el esquema de dos fases propuesto por AC4 [89] (fase de inicialización y fase de propagación) pero evita realizar, tanto el costoso chequeo, como el costoso almacenamiento que supone realizar la búsqueda de todos los soportes para todos los valores de todas las variables; por lo que AC6 se limita a buscar un solo soporte $b \in D_j$ (el más pequeño) para cada valor $a \in D_i$ que satisfaga la restricción R_{ij}. Cuando un valor de soporte b es encontrado, es almacenado en una matriz S el valor a conjuntamente con la variable X_i, como uno de los valores soportados por la variable X_j con el valor de b. Si b es eliminado del dominio de X_j, AC6 busca el siguiente soporte (más pequeño) para cada uno de los valores almacenados en la lista de soportes de S para la variable X_j. Las complejidades espacial y temporal de AC6 son $O(ed)$ y $O(ed^2)$ respectivamente, donde e es el número de restricciones y d la talla máxima del dominio.

AC6 requiere que los dominios de las variables estén ordenados de forma ascendente y necesita las siguientes estructuras de datos:

- M es una matriz de Booleanos, con dimensiones $n \times d$, que permite identificar los valores que están presentes en el dominio de cada una de las variables, donde n indica el número de variables y d la talla máxima del dominio de las variables.
- S es una matriz con dimensiones $n \times d$, que almacena para cada valor, el soporte más pequeño encontrado.

- Q es una lista que contiene las tuplas variable-valor $\langle X_i, a\rangle$ eliminadas del dominio, cuya propagación de dicha eliminación no ha sido procesada todavía.

Aprovechando que existe un ordenamiento de los dominios, AC6 en la búsqueda del siguiente soporte $c \in D_j$ para el valor $a \in D_i$, sólo la realiza, cuando consigue un valor c en D_j que sea mayor al valor b con el que se inicia la búsqueda. Esto es debido a que AC6 *conoce* que valores anteriores a b no fueron soportes válidos para a, ya sea basándose en el valor b almacenado en S (ver Algoritmo 6, paso 9); o bien porque b es iniciado con un *dummy_value* (un *dummy_value* es un valor ficticio más pequeño que el menor valor del dominio de X_j), tal y como se muestra en el Algoritmo 7 en el paso 7. Si no es posible encontrar ese nuevo valor $c \in D_j$, el valor a es eliminado.

Algoritmo 6: Procedimiento AC6

```
   Datos: CSP P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es arco-consistente) o falso y P' (el cual es
              arco-inconsistente)
 1 principio
 2     InitializeAC6(P)
 3     si initial = verdadero entonces
 4         mientras Q ≠ φ hacer
 5             Seleccione y elimine ⟨X_j, b⟩ de la cola Q
 6             para cada ⟨X_i, a⟩ ∈ S[X_j, b] hacer
 7                 elimine ⟨X_i, a⟩ de S[X_j, b]
 8                 si M[X_i, a] = 1 entonces
 9                     c ← b
10                     emptysupport ←nextSupport(R_ij, a, c)
11                     si (emptysupport =verdadero) entonces
12                         Eliminar a de D_i
13                         M[X_i, a] ← 0
14                         Q ← Q ∪ ⟨X_i, a⟩
15                         si D_i = φ entonces
16                             devolver falso
17                     sino
18                         Añadir (S[X_j, c], ⟨X_i, a⟩)
19         devolver verdadero
20     sino
21         devolver falso
22 fin
```

El Algoritmo AC6 consta de dos fases: inicialización y propagación (ver Algoritmo 6). En la fase de inicialización (ver Algoritmo 7) se busca un soporte b para cada valor a de cada variable X_i sobre la restricción R_{ij} con la que esté vinculada. El soporte buscado siempre es el valor más pequeño, para lo cual llama al Algoritmo

8. Si dicho soporte existe, este es almacenado en la estructura S (ver Algoritmo 7, paso 14). Por el contrario, si no es posible encontrar dicho soporte b, el valor $a \in D_i$ es eliminado y la tupla $\langle X_i, a\rangle$ es almacenada en la lista Q para su posterior procesamiento (ver Algoritmo 7, pasos 9 al 12).

En la fase de propagación (ver Algoritmo 6, pasos 3 al 19), es extraída una tupla $\langle X_j, b\rangle$ de la cola Q para procesar la consecuencia de su eliminación. Así, para cada una de las tuplas $\langle X_i, a\rangle$ contenidas en $S[X_j, b]$, si el valor a está aún presente en el dominio de X_i (ver Algoritmo 6, paso 10), se busca un nuevo soporte $c \in D_j$ que satisfaga la restricción R_{ij}. Si no es posible, el valor a es eliminado de D_i y una tupla $\langle X_i, a\rangle$ es añadida a Q (ver Algoritmo 6, pasos 11 al 16). Por el contrario, si se encuentra un valor de $c \in D_j$ que satisfaga la restricción R_{ij}, este pasará a formar soporte de a y será recordado en S (ver Algoritmo 6, paso 18). Este proceso continuará hasta que todas las tuplas almacenadas en Q sean procesadas o cuando se torne vacío el dominio de una variable. En el primer caso, el algoritmo indica que el problema es arco-consistente y en el segundo indica que no lo es.

Algoritmo 7: Procedimiento InitializeAC6

Datos: CSP $P = \langle X, D, R\rangle$ /*R contiene las restricciones directas e inversas*/
Resultado: Variable booleana *initial* (donde *initial* =**verdadero** indica que ningún dominio se ha quedado vacío e *initial* =**falso** indica que P' es arco-inconsistente); cola Q; matrices S y M; y CSP P' actualizados.

```
1  principio
2      Q ← {}
3      S[X_j, b] ← {}   /* ∀X_j ∈ X ∧ ∀b ∈ D_j */
4      M[X_i, a] ← 1   /* ∀X_i ∈ X ∧ ∀a ∈ D_i */
5      para cada arco R_ij ∈ R hacer
6          para cada valor a ∈ D_i hacer
7              b ← dummy_value
8              emptysupport ←nextSupport(R_ij, a, b)
9              si (emptysupport =verdadero) entonces
10                 Eliminar a de D_i
11                 M[X_i, a] ← 0
12                 Q ← Q ∪ ⟨X_i, a⟩
13             sino
14                 Añadir (S[X_j, b], ⟨X_i, a⟩)
15             si D_i = φ entonces
16                 devolver initial ← falso
17     devolver initial ← verdadero
18 fin
```

Debido a que la búsqueda se limita a un sólo soporte por cada valor del dominio de cada variable, AC6 es más rápido que AC4, cuando realiza el proceso de

Algoritmo 8: Procedimiento nextSupport

Datos: Restricción R_{ij}, valor $a \in D_i$ y valor $b \in D_j$
Resultado: Variable booleana *emptySupport* (donde *emptySupport* =**verdadero** indica que no ha encontrado un valor de soporte b en el dominio D_j para el valor a y *emptySupport* =**falso** indica que si lo ha encontrado); valor b actualizado.

```
1  principio
2      si (b ≤ que valor más grande de D_j) entonces
3          emptySupport ← falso
4          b ← siguiente valor de D_j
5          mientras ⟨a, b⟩ ∉ R_ij ∧ emptySupport =falso hacer
6              si (b < que valor más grande de D_j) entonces
7                  b ← siguiente valor de D_j
8              sino
9                  emptySupport ← verdadero
10     sino
11         emptySupport ← falso
12     devolver emptySupport y b
13 fin
```

arco-consistencia (en problemas consistentes), y el hecho de mantener estructuras más ligeras que su predecesor (AC4), AC6 puede procesar problemas con mayor número de variables, dominios y restricciones. Sin embargo, AC6 es ineficiente en la realización del proceso de chequeos de consistencia, ya que cada restricción debe ser procesada en forma directa e inversa, por lo que los chequeos sobre las restricciones no lo hace bidireccionalmente. Las versiones posteriores a AC6 propuestas en los algoritmos AC6+ [34] y AC6++ [35], eliminan este problema y sirven como preámbulo al algoritmo que le sucede: AC7 [30].

Adicionalmente, pese a realizar una búsqueda de soportes contraria a AC4, sus autores lo comparan con AC4 (que realiza una búsqueda exhaustiva de soportes) y con un algoritmo de grano grueso como lo es AC3 (AC3 también limita la búsqueda de soportes a uno, pero es un algoritmo que propaga restricciones y no valores). AC6, comparandose con AC3 y AC4, resulta favorecido en lo que respecta al número de chequeos de restricciones. Cabe destacar que sus autores, en la evaluación de AC6 dentro de la búsqueda (MAC6), indican que MAC4 realiza la búsqueda en $O(1)$ mientras que MAC6 la realiza en $O(d)$ [29].

AC7

El Algoritmo AC7 [30] es propuesto utilizando una de las propiedades de otro algoritmo de los mismos autores: AC-Inferencia, y es el conocimiento de que en

restricciones binarias el soporte es bidireccional: $\langle X_i, a\rangle$ soporta $\langle X_j, b\rangle$ si y solo si $\langle X_j, b\rangle$ soporta a $\langle X_i, a\rangle$. La inferencia de soportes que proporciona la bidireccionalidad permite a AC7 reducir el número de chequeos (evitando los chequeos inversos); permite una implementación espacial eficiente y además se evitan los chequeos de valores que son aún consistentes. Por lo tanto, AC7:

- No chequeará una tupla $\langle a, b\rangle$ sobre R_{ij}, si existe un valor b' en el dominio de X_j tal que la tupla $\langle a, b'\rangle$ ha sido ya chequeada con éxito con respecto a R_{ij}, es decir, $\langle a, b'\rangle \in R_{ij}$.
- No chequeará una tupla $\langle a, b\rangle$ sobre R_{ij} si existe un valor b' en el dominio de X_j tal que la tupla $\langle a, b'\rangle$ ha sido ya chequeada con éxito con respecto a R_{ji}, es decir, $\langle a, b'\rangle \in R_{ji}$.
- No chequeará una tupla $\langle a, b\rangle$ sobre R_{ij} si $\langle a, b\rangle$ ya ha sido chequeada con éxito; o $\langle b, a\rangle$ ha sido chequeada con éxito sobre R_{ji}.

Al igual que AC6, AC7 requiere un orden total ascendente de los dominios de las variables y de las siguientes estructuras:

- M es una matriz de Booleanos, que indica la presencia o no de un valor en el dominio de la variable. El tamaño de M es $n \times d$, donde n es el número de variables y d la talla máxima del dominio.
- CS es una matriz de $n \times d \times n$, que almacena todos los valores b en el dominio de X_j para los cuales $\langle X_i, a\rangle$ es asignado como su soporte actual. Opuesto a AC6, el soporte actual no necesariamente es el más pequeño.
- $Last$ es una matriz de $n \times d \times n$, que almacena el último valor del dominio de X_j que fue comprobado que soportará a $\langle X_i, a\rangle$. De esta forma $Last$ asegura que cada valor $b \in D_j$ es menor o igual a $Last[(X_i, a, X_j]$, y que cada b en el dominio de X_j compatible con la instanciación $\langle X_i, a\rangle$ es mayor o igual que $Last[(X_i, a, X_j]$
- SSS es una pila que almacena tuplas $\langle X_i, a\rangle$, que permite propagar las consecuencias de las podas de valores. A diferencia de los demás algoritmos, la propagación de las podas de los valores es verificada lo más pronto posible, permitiendo con ello descubrir algunas inconsistencias tempranamente.

Algoritmo 9: Procedimiento AC7

```
   Datos: CSP P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es arco-consistente) o falso y P' (el cual es
              arco-inconsistente)
 1 principio
 2     SSS ← ∅
 3     para cada X_i, X_j ∈ R_ij hacer
 4         para cada a ∈ D_i hacer
 5             CS[X_i, a, X_j] ← ∅
 6             Last[X_i, a, X_j] ← dummy_value
 7             SSS ← SSS ∪ (⟨X_i, a⟩, X_j)
 8     mientras SSS ≠ ∅ hacer
 9         Seleccione y elimine (⟨X_i, a⟩, X_j) de SSS
10         si M[X_i, a] = 1 entonces
11             c ← SeekCurrentSupport(X_i, a, X_j)
12             si c ≠ ∅ entonces
13                 CS[X_j, c, X_i] ← a
14             sino
15                 ProcesarPoda(X_i, a, SSS)
16                 si D_i = ∅ entonces
17                     devolver falso
18     devolver verdadero
19 fin
```

Durante el proceso de la arco-consistencia, AC7 realiza dos operaciones principales: la búsqueda del soporte actual para un valor y el procesamiento de la poda de un valor. Inicialmente, cada valor $\langle X_i, a\rangle$, de cada restricción R_{ij}, es colocado en la tupla $[\langle X_i, a\rangle, X_j]$ y almacenado en SSS. También son inicializados CS y $Last$ (ver Algoritmo 9, pasos 2 al 7). Luego en un bucle (pasos 8 al 17), son procesadas las tuplas almacenadas en SSS. Así, para cada tupla, AC7 busca un soporte utilizando el Procedimiento SeekCurrentSupport, de forma que si logra encontrar un soporte válido para $\langle X_i, a\rangle$, almacena en CS el soporte inverso (ver Algoritmo 9, paso 13). En caso contrario, elimina el valor a de D_i (ver Algoritmo 9, pasos 14 al 17).

Cuando el Procedimiento SeekCurrentSupport (ver Algoritmo 10) es invocado, lo primero que hace es verificar en CS la existencia en D_j de los soportes b que estén almacenados. De conseguirlo, devuelve el nuevo soporte sin haber realizado ningún chequeo, con lo que AC7 *hizo inferencia* (ver Algoritmo 10, pasos 2 al 10). Si no es posible inferir un soporte, AC7 busca a partir del valor más pequeño almacenado en $Last$ el siguiente valor b que sea consistente con R_{ij}. A diferencia de AC6, el algoritmo AC7 antes de realizar el chequeo, averigua si existe un valor de soporte en el CS inverso (ver Algoritmo 10, paso 16). De ser cierto, el valor de soporte

Algoritmo 10: Procedimiento SeekCurrentSupport

Datos: Variables $X_i, X_j \in X$; restricción $R_{ij} \in R$, valor $a \in D_i$ y matriz *Last*
Resultado: **verdadero** y valor $b \in D_j$ o **falso** cuando no encuentra ningún soporte

```
1  principio
2      encontrado ← falso
3      mientras CS[X_i, a, X_j] ≠ ∅ ∧ ¬encontrado hacer
4          b ← un elemento de CS[X_i, a, X_j]
5          si b ∈ D_j entonces
6              encontrado ←verdadero
7          sino
8              Eliminar b de CS[X_i, a, X_j]
9      si encontrado =verdadero entonces
10         devolver verdadero y b
11     b ← Last[X_i, a, X_j]
12     si b > valor más grande de D_j entonces
13         devolver falso y b = ∅
14     b ← SiguienteValor(b, D_j)
15     mientras b ≠ ∅ ∧ ¬encontrado hacer
16         si (Last[X_j, b, X_i] ≤ a) ∧ (⟨a, b⟩ ∈ R_ij) entonces
17             encontrado ← verdadero
18         sino
19             b ← SiguienteValor(b, D_j)
20     Last[X_i, a, X_j] ← b
21     devolver encontrado y b
22 fin
```

será obtenido por inferencia, de lo contrario será necesario realizar el chequeo de la restricción con los valores del dominio D_j (pasos 15 al 19) hasta lograr un soporte ($encontrado = verdadero$) o que se determine que no existe ($encontrado = falso$). El soporte encontrado b será almacenado en *Last* y el Procedimiento SeekCurrentSupport devolverá el valor que tenga almacenado en la variable *encontrado*.

Ejemplo: Sea un problema de coloreado de mapas, con dos países y tres colores disponibles : azul (a), blanco (b) y carmín (c), con la restricción de que cada país debe tener un color diferente. En la codificación de CSP, los países serán las variables X_i y X_j, los dominios serán los colores a, b y c y la restricción será: $R_{ij} : X_i \neq X_j$. La Figura 2.10 [30] muestra que: AC4 realizó 18 chequeos de restricciones, AC3 y AC6 realizaron 8 chequeos de restricciones mientras que AC7, utilizando la bidireccionalidad, sólo necesitó realizar 5 chequeos de restricciones. En dicha figura, las puntas de flecha rellenadas de negro indican que se realizó un chequeo de restricciones, las puntas de flecha sin relleno indican que se realizó inferencia. Una flecha con línea discontinua significa que el chequeo fue inconsistente con la restricción y una flecha con línea continua indica que el chequeo fue consistente. El

sentido de las flechas indican el sentido del chequeo.

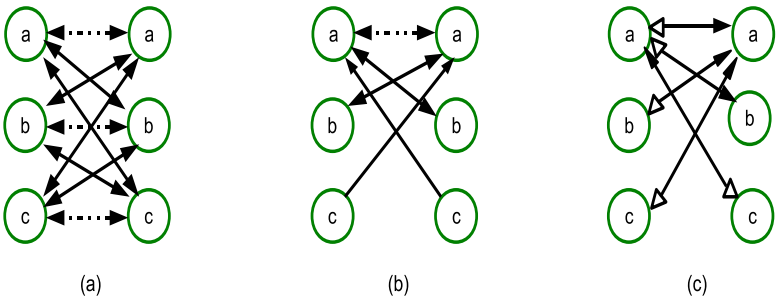


Figura 2.10: Chequeos e inferencias realizadas por los algoritmos de arco-consistencia: (a) AC4; (b) AC3 y AC6; (c) AC7 en el problema de coloreado de mapas con dos variables X_i y X_j, $D_i = D_j = \{a, b, c\}$ y la restricción $R_{ij} : X_i \neq X_j$.

AC7 tiene una complejidad espacial de $O(ed)$ y una complejidad temporal de $O(ed^2)$, donde e es el número de restricciones y d es la talla máxima de dominio. Ciertamente, AC7 es el algoritmo de grano fino que realiza menor cantidad de chequeos de restricciones, pero su eficiencia en evitar chequeos de restricciones hace que deba realizar procesos de búsqueda en la inferencia de soportes, lo cual en la práctica podría ralentizar el algoritmo.

AC2001/3.1

Tras 20 años de investigación en los algoritmos de grano-fino, resurge el interés de los investigadores por los algoritmos de grano-grueso, específicamente por el algoritmo AC3, un algoritmo muy voraz y rápido, donde su estructura de datos es muy sencilla (una cola Q) pero ineficiente, ya que al propagarse se repiten los mismos cálculos.

Considerando el marco de AC3, dos grupos de investigadores por separado desarrollan un nuevo algoritmo: AC2001/3.1 [32, 33] (ver Algoritmo 11), añadiendo una estructura de datos que almacena el soporte encontrado y continúa la búsqueda a partir de este valor.

El algoritmo AC2001/3.1 también requiere la inicialización del vector *Last* en un valor *dummy_value*, el cual que debe ser más pequeño que $minD(X_j)$ (ver Algoritmo

12). También, se modifica el procedimiento *Revise* (ver Algoritmo 13). AC2001/3.1 asume que los dominios de las variables están ordenados en forma ascendente.

Algoritmo 11: Procedimiento AC2001/3.1

```
   Datos: CSP, P = ⟨X, D, R⟩
   Resultado: verdadero y un CSP P' (el cual es arco-consistente) o falso y P' (el cual es
              arco-inconsistente porque algún dominio se ha quedado vacío)
 1 principio
 2     Initialize2001(P, Last)
 3     para cada arco R_ij ∈ R hacer
 4         Añadir (Q, R_ij)
 5         Añadir (Q, R'_ji)
 6     mientras Q ≠ φ hacer
 7         Seleccione y elimine R_ij de la cola Q
 8         si Revise2001(R_ij) = verdadero entonces
 9             si D_i ≠ φ entonces
10                 Añadir (Q, (R_ki)) con k ≠ i, k ≠ j
11             sino
12                 devolver falso /*dominio vacío */
13     devolver verdadero
14 fin
```

Algoritmo 12: Procedimiento Initialize2001

```
   Datos: CSP, P = ⟨X, D, R⟩.
   Resultado: Matriz Last inicializada con valores dummy_value.
 1 principio
 2     para cada R_ij ∈ R hacer
 3         para cada a ∈ D_i hacer
 4             Last(R_ij, X_i, a) ← dummy_value
 5         para cada b ∈ D_j hacer
 6             Last(R_ij, X_j, b) ← dummy_value
 7 fin
```

El Procedimiento Revise2001 (ver Algoritmo 13) verifica para cada uno de los valores a de D_i, antes de buscar su soporte, que el valor almacenado en la matriz $Last(R_{ij}, X_i, a)$ esté presente en el dominio D_j. De ser cierto, evita realizar chequeos inefectivos, puesto que ese valor a ya fue previamente verificado y continúa soportado; por lo que procede con el siguiente valor. En caso de que $Last(R_{ij}, X_i, a)$ contenga un valor *dummy_value* o el valor que tenga ya no forme parte de D_j, procede a buscar un valor $b \in D_j$ que satisfaga la restricción R_{ij}. Si no encontra ese valor b, elimina el valor $a \in D_i$, por no estar soportado en D_j y asigna el valor de verdadero a la variable Booleana *change*.

La forma de procesar de AC2001/3.1 es la base de los algoritmos de grano-grueso

Algoritmo 13: Procedimiento Revise2001

Datos: CSP P' definido por dos variables $X = \{X_i, X_j\}$, dominios $D = \{D_i, D_j\}$, y una restricción $R = \{R_{ij}\}$.
Resultado: D_i, tal que X_i es arco-consistente relativo a X_j y la variable booleana *change* (donde *change* = **verdadero** indica que el dominio D_i ha sido podado y *change* = **falso** indica que el dominio no se ha modificado).

```
1  principio
2      change ← falso
3      para cada a ∈ D_i hacer
4          si valor almacenado en Last(R_ij, X_i, a) ∉ D_j entonces
5              si ∄b ∈ D_j tal que (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ R_ij entonces
6                  Eliminar a de D_i
7                  change ← verdadero
8              sino
9                  Last(R_ij, X_i, a) ← b
10     devolver change
11 fin
```

que le suceden: AC3.2, AC3.3; AC3rm, etc. AC2001/3.1 es el único algoritmo de grano-grueso óptimo. Su complejidad espacial es $O(ed^2)$ y su complejidad temporal es $O(ed)$, donde e es el número de restricciones y d es la talla máxima del dominio. AC2001/3.1 es eficiente en cuanto a la re-revisión de soportes, ya que evita su chequeo si ellos aún conservan su soporte. No obstante, es ineficiente ya que no actualiza el chequeo bidireccionalmente (este problema lo resuelve el algoritmo AC3.3 [76]) e igual que el resto de algoritmos de arco-consistencia, siempre propaga cuando poda valores, sin considerar si dicha propagación va a ser efectiva o no.

AC3r y AC3rm

Los algoritmos AC3r y AC3rm [77] son propuestos para optimizar el algoritmo AC3, explotando lo que sus autores denominan *residuos de soportes*. Un *residuo de soporte* o simplemente residuo, es un soporte que ha sido almacenado durante una ejecución previa del procedimiento, que determina si un valor es soportado por una restricción. El punto clave es que un residuo no garantiza representar el límite inferior del soporte actual más pequeño de un valor. Así, el algoritmo AC3 es refinado explotando los residuos de la siguiente forma: antes de iniciar la búsqueda *desde cero* del soporte de un valor, verifica la validez del residuo asociado con el valor. Este es el funcionamiento básico de AC3r y cuando se le añade la multidireccionalidad (bidireccionalidad, en restricciones de dos variables) se obtiene el algoritmo AC3rm.

Durante el proceso de arco-consistencia, AC3rm almacena en una cola Q, tuplas compuestas por restricción y variable. Así, cada restricción $R_{ij} \in R$ proporcionará dos tuplas (R_{ij}, X_i) y (R_{ij}, X_j), ya que la restricción es binaria. El residuo o último soporte encontrado t es almacenado en una matriz $supp[R_{ij}, X_s, a]$.

Algoritmo 14: Procedimiento AC3rm

```
   Datos: CSP, P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es arco-consistente) o falso y P' (el cual es
              arco-inconsistente porque algún dominio se ha quedado vacío)
 1 principio
 2     Q ← (R_ij, X_i) ∧ (R_ij, X_j) | R_ij ∈ R ∧ X_i, X_j ∈ X
 3     mientras Q ≠ φ hacer
 4         Seleccione y borre (R_ij, X_s) de la cola Q con X_s = X_i ∨ X_s = X_j
 5         si Revise3rm((R_ij, X_s)) = verdadero entonces
 6             si D_i ≠ φ entonces
 7                 Q ← (R_ks, X_k) | R_ks ≠ R_ij, X_k ≠ X_s
 8             sino
 9                 devolver falso /*domino vacio*/
10     devolver verdadero
11 fin
```

El Procedimiento AC3rm se inicia guardando las tuplas $\langle R_{ij}, X_i\rangle$ y $\langle R_{ij}, X_j\rangle$ en la cola Q (ver paso 2, del Algoritmo 14). Después, en un bucle, es seleccionado y revisado un arco de Q (pasos 4 y 5). Si el proceso de revisión ha efectuado poda, se actualiza la cola Q con los arcos que requieren re-revisión (paso 7). El algoritmo finaliza cuando se vacía un dominio, o la cola Q.

Cada revisión realizada por Revise3rm (ver Algoritmo 15) supone verificar para cada valor la validez del último soporte encontrado. Si esto falla, se empieza una búsqueda *desde cero* de un nuevo soporte con la función SeekSupport (ver Algoritmo 16). Notación: el símbolo $\perp$ denota el mínimo valor del dominio de la variable; el símbolo $\top$ denota el hecho de que se hayan revisado y agotado el dominio de la variable; y $t[X_i] = a_i$ denota al valor que pertenece a D_i.

El Algoritmo AC3r utiliza algoritmos y estructuras similares a AC3rm, pero como no explota la bidireccionalidad, no guarda simétricamente los soportes encontrados, por lo que en la codificación de la Función Revise3r hay que sustituir las líneas 7 y 8 del Algoritmo 15 por la siguiente sentencia: $supp[C, X_s, a] \leftarrow t$. El algoritmo AC3rm admite un comportamiento óptimo en problemas con alta restringibilidad en las restricciones. AC3r y AC3rm tienen una complejidad espacial $O(ed)$

Algoritmo 15: Función Revise3rm

```
Datos: Tupla (R_ij, X_s).
Resultado: verdadero cuando ha realizado poda en D_i o falso cuando no se ha modificado el D_i.
1  principio
2      nbElements ← |D_s| /* guardar el D_S */
3      para cada a ∈ D_s hacer
4          si supp[R_ij, X_s, a] es válido entonces
5              continue
6          sino
7              t ← SeekSupport(R_ij, X_s, a)
8              si t = ⊤ entonces
9                  Eliminar a de D_i
10             sino
11                 para cada S_y ∈ R_ij hacer
12                     supp[R_ij, S_y, t[S_y]] ← t /*bidireccionalidad*/
13     devolver nbElements ≠ |D_s|
14 fin
```

Algoritmo 16: Función SeekSupport

```
Datos: Tupla (R_ij, X_s, a).
Resultado: Valor t válido o valor t = dummy_value (se ha agotado el dominio de valores para la
           variable)
1  principio
2      t ← ⊥ /* guardar el min valor de D_S */
3      mientras t ≠ ⊤ hacer
4          si t es válido en R_ij entonces
5              devolver t
6          t ← SetNextTuple((R_ij, X_s, a, t))
7      devolver ⊤
8  fin
```

y temporal $O(ed^3)$, donde e es el número de restricciones y d es la talla máxima del dominio.

2.3.3. Arco Consistencia Generalizada

La definición de arco-consistencia ha sido extendida a CSPs no-binarios. Normalmente, a esta consistencia se le llama arco-consistencia generalizada GAC [89, 28].

Definición 2.3.6. Un CSP no-binario cumple la *consistencia de arco generalizada*, si para cada variable X, que está involucrada en una restricción k-aria, todos los valores en el dominio de X tienen al menos una tupla soporte en la restricción k-aria.

Ejemplo: Consideramos la restricción no-binaria "$X_1 + X_2 + X_3 + X_4$ es par",

donde los dominios de las variables X_1, X_2, X_3 y X_4 son $\{1\}, \{0\}, \{0\}$ y $\{0, 1\}$ respectivamente. Esta restricción requiere que la suma de las cuatro variables sea par. El valor 0 de D_4 no tiene una tupla soporte en la restricción, ya que la tupla $\{\langle X_1, 1\rangle, \langle X_2, 0\rangle, \langle X_3, 0\rangle, \langle X_4, 0\rangle\}$ no es válida, por lo que la consistencia de arco generalizada eliminará el valor 0 del dominio de la variable X_4.

2.3.4. Arco-consistencia direccional

Un caso particular de arco-consistencia es la arco-consistencia direccional (DAC). Este tipo de arco-consistencia es usado por algoritmos que asignan los valores a las variables siguiendo un orden dado. Ello solo exige que para cualquier valor $a \in D_i$ de cualquier variable X_i, y para todos los arcos dirigidos R_{ij}, exista un valor b en el dominio D_j que satisfaga la restricción (R_{ij}). De este modo, la arco-consistencia direccional sólo sirve para garantizar que los valores de las variables de menor orden son consistentes con los valores de las variables de mayor orden.

2.3.5. Consistencia de Senda (Path-consistencia)

La path-consistencia (consistencia de senda) [28] es un nivel más alto de consistencia local que la arco-consistencia. La path-consistencia requiere para cada par de valores a y b de dos variables X_i y X_j, que las instanciaciones $\langle X_i, a\rangle$ y $\langle X_j, b\rangle$ que satisfacen la restricción entre X_i y X_j, que 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. Cuando un problema satisface la path-consistencia y además es nodo-consistente y arco-consistente, se dice que satisface fuertemente la path-consistencia (strongly path-consistent).

Definición 2.3.7. Un par de variables (X_i, X_j) es *path-consistente* si y solo si $\forall (a, b) \in R_{ij}, \forall X_k \in X, \exists c \in D_k$ tal que c está soportado por a en R_{ik} y c está soportado por b en R_{jk}.

Definición 2.3.8. Un CSP binario es *path-consistente* si y solamente si $\forall X_i, X_j \in X, (X_i, X_j)$ es path-consistente.

El estudio de la path-consistencia no es ampliamente utilizado debido a que presenta un alto coste temporal y espacial. Además puede modificar el grafo de restricciones añadiendo nuevas restricciones binarias. Sin embargo, es una importante consistencia en CSPs con ciertas propiedades como los CSPs temporales [43].

2.3.6. Consistencias Singulares

Debruyne y Bessiere introdujeron *consistencias singulares* como una nueva clase de consistencias [40]. Estas consistencias están basadas en el hecho de que si un valor a de una variable X_i es consistente, entonces el CSP obtenido mediante la asignación de a a X_i es consistente. Se denotará con $P_{D_i=a}$ al CSP obtenido al restringir D_i al valor a en un CSP P.

Para un valor a de la variable X_i, una consistencia singular examinará si es consistente o no aplicando una consistencia local a $P_{D_i=a}$. Si $P_{D_i=a}$ es inconsistente, entonces a es eliminado de D_i.

2.3.7. Consistencias Inversas

La idea de *consistencia inversa* fue introducida por Freuder y Elfe [50]. Esta consistencia elimina valores de las variables que no son consistentes con cualquier instanciación consistente, de un conjunto dado de variables adicionales. Las consistencias inversas pueden ser definidas en los términos de (i, j)-consistencias. Cuando i es 1 y j es $k-1$ tenemos la k inversa consistencia. Las consistencias donde $i > 1$, como la path-consistencia, identifican y almacenan combinaciones de valores para las variables para las cuales no se puede encontrar un valor consistente para algunas variables adicionales. Esto es, estas consistencias añaden restricciones no unarias al problema. Esto lleva a cabo incrementos en el coste temporal que pueden llegar a ser prohibitivos. La ventaja de las consistencias inversas es que ellas sólo eliminan valores de las variables, sin la necesidad de añadir restricciones no unarias, con el correspondiente ahorro de espacio.

La consistencia inversa más simple es la *consistencia de senda inversa*[2], que es equivalente a $(1, 2)$-consistencia. De acuerdo a la definición de consistencia de senda inversa, un valor a de una variable x cumple la consistencia de senda inversa si ese

[2]Arco-consistencia inversa es equivalente a $(1, 1)$-consistencia, es decir, arco-consistencia

valor puede ser extendido de forma consistente a todas las ternas de variables que contienen a x. Sin embargo, Debruyne y Bessiere demostraron que no es necesario chequear esta condición, para los valores con el objetivo de alcanzar la consistencia de senda inversa.

Definición 2.3.9. Un CSP cumple con la *consistencia de senda inversa*, si y solo si es arco-consistente y para cada valor $a \in D_i$, y para cada terna de tres variables (X_i, X_j, X_k), la instanciación $\langle X_i, a\rangle$ puede ser extendida a una instanciación consistente de la terna (X_i, X_j, X_k).

2.3.8. Consistencia de borde

Algunas veces comprobar la consistencia de arco o cualquier otro nivel de consistencia más fuerte puede resultar demasiado costoso cuando los dominios de las variables del problema son muy grandes o para problemas con dominios continuos donde los valores son números reales o decimales. La consistencia de borde es una aproximación de la consistencia de arco la cual solo requiere chequear la arco-consistencia al límite superior e inferior del dominio de la variable [31, 96]. La consistencia de borde es una noción genérica que puede ser aplicada a CSPs de cualquier aridad. Sin embargo la consistencia de borde es más débil que la consistencia de arco.

Ejemplo: Consideremos un problema con dos variables X, Y con dominios $\{1, ..., 8\}$ y $\{-2, ..., 2\}$, respectivamente y la restricción $X = Y^3$. Este problema cumple la consistencia de borde, ya que los valores $1, 8$ para X y $-2, 2$ para Y son consistentes con la restricción. Sin embargo, el problema no es arco-consistente porque la instanciación $\langle Y, 0\rangle$ es inconsistente, ya que no hay valores en D_X que satisfagan la restricción cuando $\langle Y, 0\rangle$.

2.4. Técnicas de Búsqueda

Las técnicas de búsqueda en los CSPs se pueden subdividir en técnicas completas y en técnicas incompletas o locales, como se muestra en las Figuras 2.11 y 2.12. Los algoritmos completos o sistemáticos buscan a través del árbol de búsqueda las

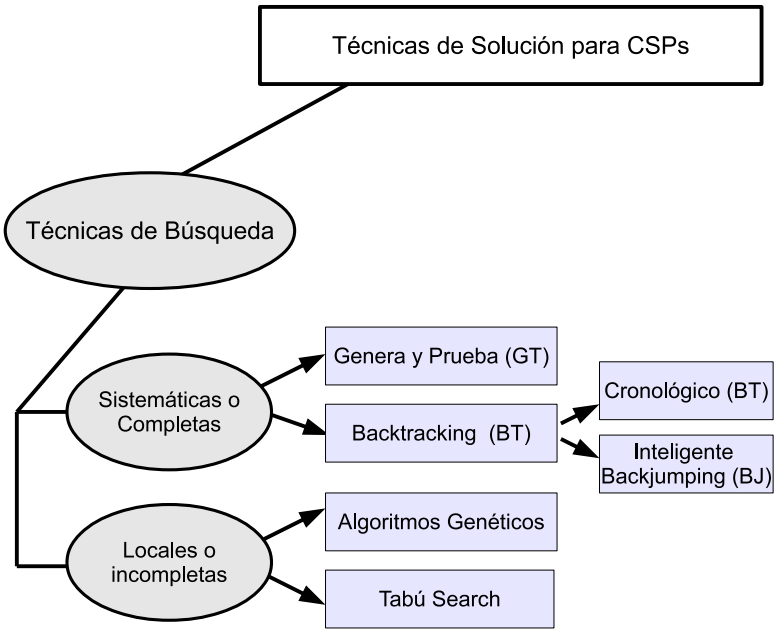


Figura 2.11: Clasificación de las técnicas de búsqueda de solución para CSPs.

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. El algoritmo de *Backtracking* [37] es la base fundamental de los algoritmos de búsqueda sistemática para la resolución de CSPs.

Los algoritmos incompletos también son muy utilizados en problemas de satisfacción de restricciones y en problemas de optimización. Pese a que no garantizan encontrar una solución, son mucho más rápidos que los algoritmos completos, ya que incorporan heurísticas en el proceso de resolución.

Así, se han desarrollado muchos algoritmos de búsqueda completa para CSPs. Algunos ejemplos son: 'backtracking cronológico' [37], 'backjumping' [62], y algoritmos híbridos como: 'conflict-directed backtracking' [94], 'backtracking dinámico' [69], 'forward-checking' [70], 'minimal forward checking' [44], forward checking con conflict-directed backtracking [94], sobre los cuales se hará una breve explicación.

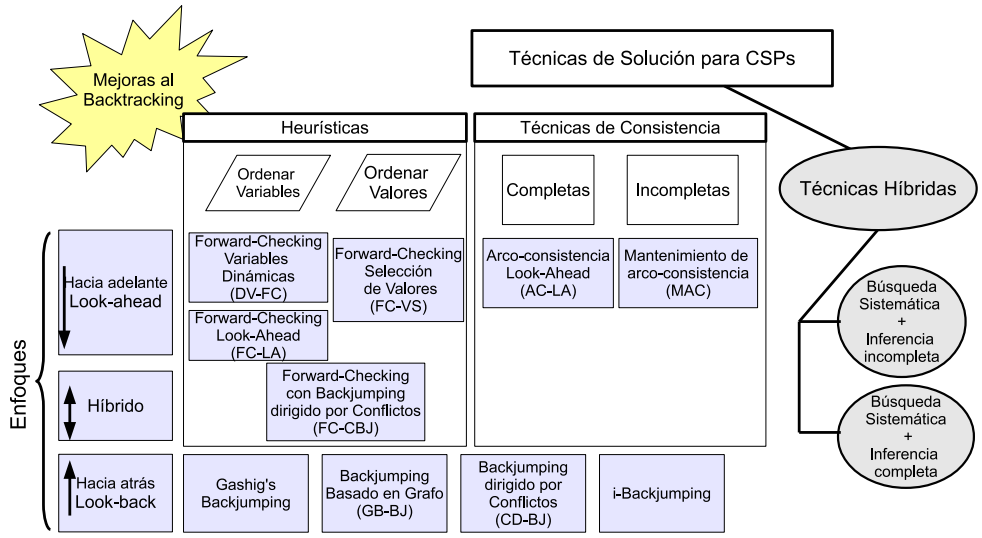


Figura 2.12: Clasificación de las técnicas de solución híbridas para CSPs.

2.4.1. Genera y Prueba

El algoritmo Genera y Prueba (generate-and-test) es la técnica de búsqueda completa más sencilla que permite encontrar todas las soluciones a un CSP. El algoritmo genera de forma sistemática todas las asignaciones completas posibles y una vez terminada la asignación completa, comprueba si ésta satisface todas las restricciones. La ineficiencia de este algoritmo consiste en que genera muchas asignaciones completas que violan la misma restricción.

2.4.2. Backtracking Cronológico

El algoritmo básico de búsqueda sistemática para resolver CSPs es el *Algoritmo de Backtracking Cronológico* [37, 78], o simplemente *Backtracking* (BT), que permite mejorar la ineficiencia del algoritmo Genera y Prueba. Se asume que: (a) el CSP es binario; y (b) la existencia de un orden estático entre las variables y entre los valores de los dominios de dichas variables.

BT realiza el trabajo de búsqueda utilizando dos procesos: hacia adelante y hacia atrás. En el proceso hacia adelante, el algoritmo selecciona la siguiente variable de acuerdo al orden de las mismas y le asigna su próximo valor. Esta asignación de

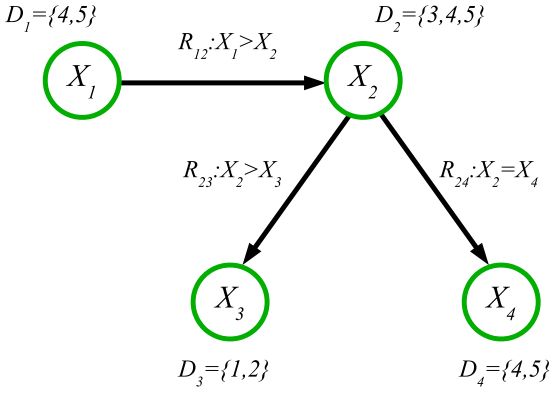


Figura 2.13: Ejemplo de CSP binario normalizado con cuatro variables.

la variable se verifica en todas las restricciones formadas por la variable actual y alguna de las variables anteriores. Si todas las restricciones formadas por variables ya asignadas se han satisfecho, el algoritmo selecciona la siguiente variable y trata de encontrar un valor para ella de la misma manera. Si alguna restricción en la que participa esta variable no se satisface, entonces se activa el proceso hacia atrás: 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 la asignación de un nuevo valor. Si asumimos que estamos buscando una sola solución, BT 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.

Ejemplo: La Figura 2.13 muestra el grafo de un CSP con cuatro variables [85] y tres restricciones. Cuando es procesado por BT, recorre el árbol de búsqueda como muestra la Figura 2.14.

BT es un algoritmo muy simple pero ineficiente. El problema es que tiene una visión local del problema. Sólo verifica 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 no 'recuerda' las acciones previas, y como resultado, puede repetir la misma acción varias veces innecesariamente.

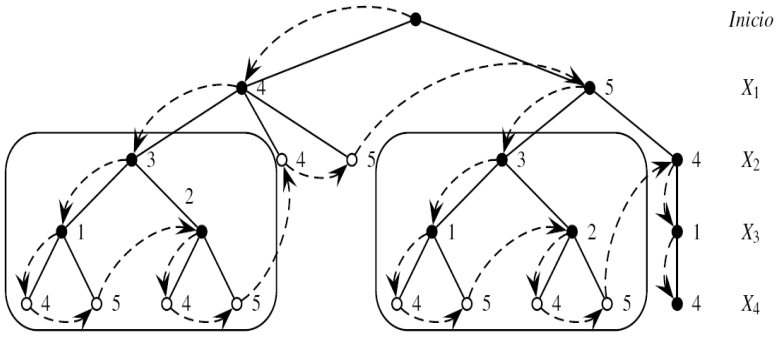


Figura 2.14: Recorrido del árbol de búsqueda utilizando BT para el problema de la Figura 2.13.

Para ayudar a combatir los problemas del algoritmo BT, se han desarrollado algunos algoritmos de búsqueda más eficientes. La forma de trabajo de estos algoritmos se puede dividir en tres enfoques: Hacia atrás (look-back), hacia adelante (look-ahead) e híbridos (combinación de los enfoques anteriores, los cuales serán ampliados a continuación (ver esquema de Figura 2.12).

2.4.3. Técnicas con enfoque Look-Back

Los algoritmos con enfoque 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 el chequeo de la consistencia *hacia atrás*, es decir, entre la variable actual y las pasadas. A continuación, presentamos algunos algoritmos que utilizan el enfoque look-back:

- *Backjumping*(BJ) [62] es un algoritmo para CSPs parecido al backtracking cronológico, con la diferencia 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 (más cerca de la raíz, en el árbol de búsqueda) 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

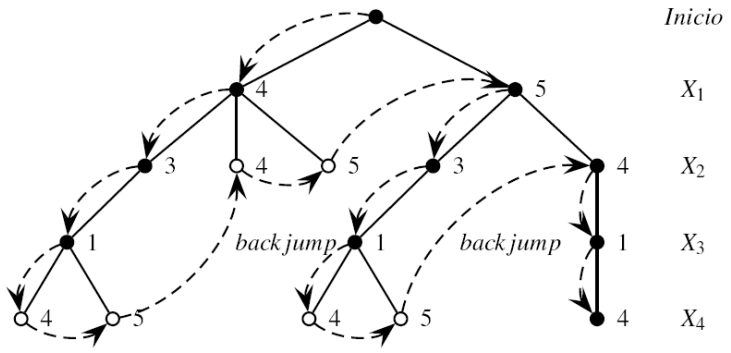


Figura 2.15: Recorrido del árbol de búsqueda utilizando BJ para el problema de la Figura 2.13.

(debido a la restricción entre X_j y X_i). Cambiar la instanciación de X_j permite encontrar una instanciación consistente de la variable actual.

El recorrido en el árbol de búsqueda que realiza BJ para el ejemplo de la Figura 2.13, es mostrado en la Figura 2.15.

- *Conflict-directed Backjumping*(CBJ) [94] tiene un comportamiento de salto hacia atrás más sofisticado que BJ. Cada variable X_i tiene un *conjunto de conflictos* formado por las variables pasadas que están en conflicto con X_i. En el momento en el que el chequeo 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 complejas para almacenar los conjuntos conflictos.

- *Learning* [51] 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, sabemos que la tupla de asignaciones $(X_1, a_1), ..., (X_{i-1}, a_{i-1})$ nos lleva a una situación sin salida. Así, podemos *aprender* que una combinación de asignaciones para

las variables $X_1, ..., X_{i-1}$ no está permitida.

2.4.4. Técnicas con enfoque Look-Ahead

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, en todos los casos, el chequeo de la consistencia se realiza solamente hacia atrás, ignorando las futuras variables.

Los algoritmos con enfoque *Look-ahead* hacen un chequeo hacia adelante en cada etapa de la búsqueda, es decir, ellos chequean para obtener las inconsistencias de las variables futuras involucradas, además de las variables actual y pasadas. De esta manera, las situaciones sin salida se pueden identificar antes y además, los valores inconsistentes se pueden descubrir y podar en las variables futuras.

Forward Checking

- *Forward checking* (FC) [70] es uno de los algoritmos look-ahead más comunes. En cada etapa de la búsqueda, FC chequea 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 *backtrack*. FC garantiza que en cada etapa la solución parcial actual sea consistente con cada valor de cada variable futura. Así, cuando se asigna un valor a una variable, chequea la consistencia con las variables futuras con las que está involucrada. Así, mediante el chequeo hacia adelante, FC puede identificar antes las situaciones sin salida y podar el espacio de búsqueda. El proceso de FC se puede ver como la aplicación de un simple paso de arco-consistencia direccional sobre cada restricción que involucra la variable actual con una variable futura, después de cada asignación de valor a una variable.

  Ejemplo: La Figura 2.16 muestra a la izquierda el problema de las 4 reinas, su representación mediante grafos y en (a) el árbol de búsqueda que genera

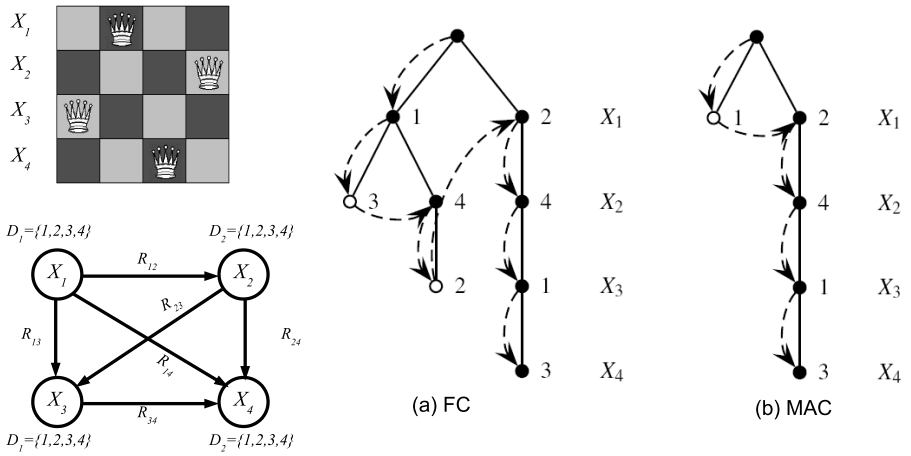


Figura 2.16: Problema de 4 reinas, grafo de representación y solución mediante dos técnicas: (a) árbol de búsqueda generado por FC; (b) árbol de búsqueda generado por MAC.

cuando realiza FC.

Forward checking se ha combinado con algoritmos look-back para generar algoritmos *híbridos* [94]. Por ejemplo, *forward checking* con *conflict-directed backjumping* (FC-CBJ) [94] es un algoritmo híbrido que combina el movimiento hacia adelante de FC, con el movimiento hacia atrás de CBJ. De esa manera, posee las ventajas de ambos algoritmos.

- *Minimal forward checking* (MFC) [44] es una versión de FC que pospone todo el chequeo de la consistencia de FC hasta que sea absolutamente necesario. En vez de chequear hacia adelante la asignación actual contra todas las variables futuras, MFC sólo chequea si la asignación actual causa una poda de dominios. Para hacer esto, es suficiente con chequear la asignación actual contra los valores de cada variable futura, hasta que se encuentra una que sea consistente. Después, si el algoritmo ha retrocedido, vuelve atrás y lleva a cabo los chequeos *perdidos*. Claramente, MFC siempre lleva a cabo a lo sumo el mismo número de chequeos que FC. Resultados experimentales han demostrado que la ganancia no supera el 10 % [44].

- Nuevas versiones de FC han sido propuestas [20]. Estas técnicas 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 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 de conflictos y los usan para podar los valores a los niveles pasados de búsqueda. Una diferencia con CBJ es que los conjuntos de conflictos se almacenan sobre un valor y no sobre una variable, es decir, cada valor de cada variable tiene su propio conjunto de conflictos. Esto permite saltar más lejos que CBJ.

2.4.5. Mantenimiento de la Arco-consistencia

Los algoritmos de búsqueda 'manteniendo la arco-consistencia' (MAC)[3], combinan un algoritmo de búsqueda con una técnica de inferencia incompleta: arco-consistencia [36]. Estos algoritmos son actualmente los más utilizados en los resolutores de CSP[4]. Los algoritmos MAC realizan más trabajo cuando miran hacia adelante, tras una instanciación o eliminación de valores [62, 104]. Cuando se intenta la asignación de una variable, MAC aplica arco-consistencia al sub-problema formado por todas las futuras variables. Esto significa que además del trabajo que FC realiza, MAC chequea la consistencia también las variables pasadas. Si el dominio de alguna variable futura se queda vacío, la instanciación realizada a la variable actual se deshace, el valor probado se elimina del dominio de la variable actual y se prueba la arco-consistencia de nuevo. Después se prueba con un nuevo valor de

[3]Inicialmente el nombre fue Real-Full-Look-Ahead (RFLA)

[4]En http://www.cril.univ-artois.fr/CPAI08/Competition-08.pdf están las especificaciones de diseño de los resolutores que utilizan MAC.

la variable actual. Si ya no quedan valores en el dominio de la variable actual, entonces, al igual que en FC, se lleva a cabo el backtracking. El trabajo extra que MAC realiza al aplicar arco-consistencia puede eliminar más valores de las futuras variables y como consecuencia logra podar más el árbol de búsqueda que FC.

Ejemplo: La Figura 2.16 muestra a la izquierda el problema de las 4 reinas, su representación mediante grafos y en (b) el árbol de búsqueda que genera cuando realiza MAC. Como puede observarse tiene un mejor desempeño que FC.

Dependiendo del algoritmo de arco-consistencia que se utilice en el algoritmo MAC se generan diferentes versiones del mismo. En [77], realizan un estudio comparativo entre los algoritmos MAC que se listan a continuación, donde MAC3 es el que tiene el más bajo rendimiento:

- MAC3 mantiene arco-consistencia utilizando el algoritmo AC-3,
- MAC2001 usa el algoritmo AC2001/3.1,
- MAC3.2 usa el algoritmo AC3.2,
- MACrm usa el algoritmo AC3rm.

El algoritmo MAC, al igual que el algoritmo FC, se ha combinado con algoritmos con enfoque look-back para obtener algoritmos híbridos. Por ejemplo MAC-CBJ [95] mantiene arco-consistencia cuando se mueve hacia adelante y utiliza el conjunto de conflictos de CBJ para saltar hacia atrás cuando se mueve hacia atrás. En [20], donde se propusieron extensiones de FC, hay también una extensión de MAC, llamada *conflict based MAC* (CFMAC). Al igual que CFFC, CFMAC utiliza valores basados en conjuntos de conflictos para hacer más podas y alcanzar saltos hacia atrás más grandes.

Para trabajar con restricciones no-binarias, el algoritmo MAC se ha generalizado a un algoritmo que mantiene arco-consistencia generalizada (MGAC). Generalmente, la arco-consistencia generalizada sólo se mantiene sobre restricciones para las cuales existen algunos algoritmos especializados de filtrado de baja complejidad, como por ejemplo, restricciones de *todos diferentes* o restricciones de cardinalidad. Por ejemplo, Régin y Puget resolvieron el problema de rutas de vehículos que incluían restricciones de secuencia utilizando un algoritmo de arco-consistencia generalizada, especializado para tales restricciones [99].

2.5. Heurísticas

Un algoritmo de búsqueda para la satisfacción de restricciones requiere del orden en el cual se van a estudiar las variables, del orden en el que se van a estudiar los valores del dominio en cada una de las variables, así como del orden en que se van a aplicar las restricciones. Seleccionar el orden correcto de las variables, valores y restricciones puede mejorar notablemente la eficiencia de resolución. Las heurísticas de ordenación de variables, de valores y de restricciones juegan un importante papel en la resolución de CSPs.

2.5.1. Heurísticas de 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 un impacto significativo en el tamaño del espacio de búsqueda explorado. Esto se debe a la gran cantidad de trabajo que se ha llevado a cabo, tanto en las heurísticas de ordenación de variables estáticas como dinámicas.

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 genera durante la búsqueda. Generalmente, las heurísticas de ordenación de variables tratan de seleccionar lo antes posible las variables que más restringen a las demás. Se intuye que es conveniente tratar de asignar lo antes posible las variables más restringidas y de esa manera identificar las situaciones sin salida lo antes posible, reduciendo así el número de backtracking.

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 *minimun width* (MW) [49] 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 una variable X_i es el número de variables que están antes de X_i, de acuerdo a un orden dado, y que son adyacentes que X_i. 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 órdenes. 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.

La heurística *maximun degree* (MD) [42] ordena las variables en un orden decreciente con respecto a 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) [97] 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 [42] se compararon varias heurísticas de ordenación de variables estáticas utilizando CSPs, generados aleatoriamente. Los resultados experimentales probaron que todos ellos son peores que MRV, que es una heurística de ordenación de variables dinámicas que presentaremos a continuación.

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. Por este motivo estas heurísticas generalmente se utilizan en algoritmos de chequeo 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 es *minimun remaining values* (MRV). MRV se basa en el principio de *primer fallo* (FF), el cual que sugiere que *para tener éxito deberíamos intentar primero donde sea más probable*

que falle [70]. De esta manera las situaciones sin salida pueden identificarse antes. Además, se ahorra espacio de búsqueda. De acuerdo con el principio de FF, en cada paso, se selecciona la variable más restringida. Por esta razón, MRV selecciona la variable con el dominio más pequeño. Se intuye que si una variable tiene pocos valores, entonces es más difícil encontrar un valor consistente. Cuando se utiliza MRV junto con algoritmos de chequeo 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 usa 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 por asignar es la variable con el dominio más pequeño. MRV es en general una heurística de ordenación de variables general, ya que se puede aplicar a CSPs de cualquier aridad.

En [73] se introduce una heurística que trata de seleccionar la variable más restringida evaluando la restringibilidad de cada valor de cada variable. Es decir, para cada variable X no instanciada, se evalúan los valores de X con respecto al número de valores de las futuras variables que no son compatibles (más detalles en la Sección 2.5.2). Estas evaluaciones se combinan para producir una estimación de cuan restringida está cada variable. Se selecciona la variable más restringida.

Geelen propuso una heurística similar, basada en el conteo del número de valores soporte para cada valor de una variable no instanciada [63]. Geelen llamó *promesa* de un valor a al producto de los valores que lo soportan, en todas las variables futuras. De esta manera, la heurística de ordenación de variables de Geelen mide la promesa de los valores para cada variable no asignada y selecciona la variable, cuya suma de promesas de sus valores es mínima. La heurística trata así de minimizar el número total de valores que pueden asignarse a todas las variables futuras de forma que no se viole ninguna restricción de la variable seleccionada. Un efecto lateral de la heurística de ordenación de variables de Geelen es que se asegura la arco-consistencia en cada etapa de la búsqueda, pero es una heurística de alta complejidad. Su complejidad temporal para asignar un valor a una variable es $O(n^2d^2)$ donde n es el número de variables y d es la talla del dominio más grande.

2.5.2. Heurísticas de Ordenación de Valores

En comparación a la ordenación de variables, 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.

Heurísticas *min-conflicts*

Una de las heurísticas de ordenación de valores más conocidas es la heurística *min-conflicts*. 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 [73], 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 proporciona 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.

Heurística *promise*

Geelen [63] propuso una heurística de ordenación de valores a la cual llamó *promise*. Para cada valor a de la variable X_i, contamos el número de valores que soporta a en cada futura variable adyacente 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 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, una vez el valor haya sido asignado a la variable.

Heurísticas *domain-size*

En [52], se describen tres heurísticas de ordenación de valores dinámicos: *max-domain-size*, *weighted-max-domain-size* y *point-domain-size*, inspirados por la intuición de que un subproblema es más probable que tenga solución, si no tiene variables, cuyo dominio contenga un único valor.

La primera heurística, llamada heurística *max-domain-size*, selecciona el valor de la variable actual mediante el cual se obtiene 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 tienen 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*, asigna un peso (unidades) a cada valor de la variable actual, dependiendo del número de variables futuras, cuyos dominios tengan una talla determinada. Por ejemplo, para cada variable futura cuyo dominio tenga una talla igual a 1, como consecuencia de la asignación, se añaden 8 unidades al peso de a_i. De esta manera se selecciona el valor con el menor peso. Estas heurísticas parecen ser muy a medida y los resultados experimentales [52] muestran que la heurística *min-conflict* supera a estas tres heurísticas.

2.5.3. Heurística de Ordenación de Restricciones

La heurística de ordenación de restricciones, COH (Constraint Ordering Heuristic) propuesta por Salido [107] es una heurística de pre-proceso basada en el muestreo de una población. COH puede ser aplicada a cualquier algoritmo basado

en búsqueda por backtracking y permite:

- estudiar la restringibilidad del problema, por medio del parámetro $\mathcal{T}$ que mide la restringibilidad del problema. $\mathcal{T}$ representa la probabilidad de que un problema sea factible. $\mathcal{T} = 0$ corresponde a un problema sobre-restringido donde no se esperan estados que puedan ser solución, mientras que $\mathcal{T} = 1$ corresponde a un problema poco-restringido donde cualquier estado puede ser solución.

- clasificar las restricciones no-binarias, independientemente de su aridad (cantidad de variables por restricción), de forma que las restricciones más restrictivas sean estudiadas primero.

Para ello, COH ordena las restricciones del problema basado en la restringibilidad de las mismas, para que las restricciones más restrictivas sean procesadas primero, permitiendo que las tuplas inconsistentes sean encontradas en las fases iniciales y reduciendo de forma significativa el número de chequeos de restricciones.

Para la determinación de $\mathcal{T}$ y la clasificación de las restricciones, COH requiere calcular el tamaño la población de donde tomará una muestra. Dicha población estará compuesta por los puntos situados en el borde convexo de todos los estados iniciales generados del producto cartesiano de los límites de los dominios de las variables. Así, la muestra estará compuesta por puntos $s(n, d, e)$ distribuidos al azar o por muestreo sistemático, donde s es una función que depende del número de variables (n), la talla del dominio (d) y el nivel de precisión (e), seleccionado por el usuario. La fórmula para calcular s es la siguiente:

$s(n, d, e) := \lceil \frac{d^n}{1+d^n \cdot e^2} \rceil$

De esta forma, es calculada la restringibilidad de cada una de las restricciones (tightness), tal que para cada restricción R_{ij} los puntos de muestra son proyectados en las variables contenidas en la restricción y es realizado un chequeo de consistencia. El valor de restringibilidad calculado es utilizado para etiquetar a la restricción. Las restricciones etiquetadas serán clasificadas ascendentemente según su restringibilidad (primero irán las restricciones más restrictivas y posteriormente las menos restrictivas). Finalmente el problema con este nuevo orden de restricciones será enviado al resolutor CSP para su procesamiento.

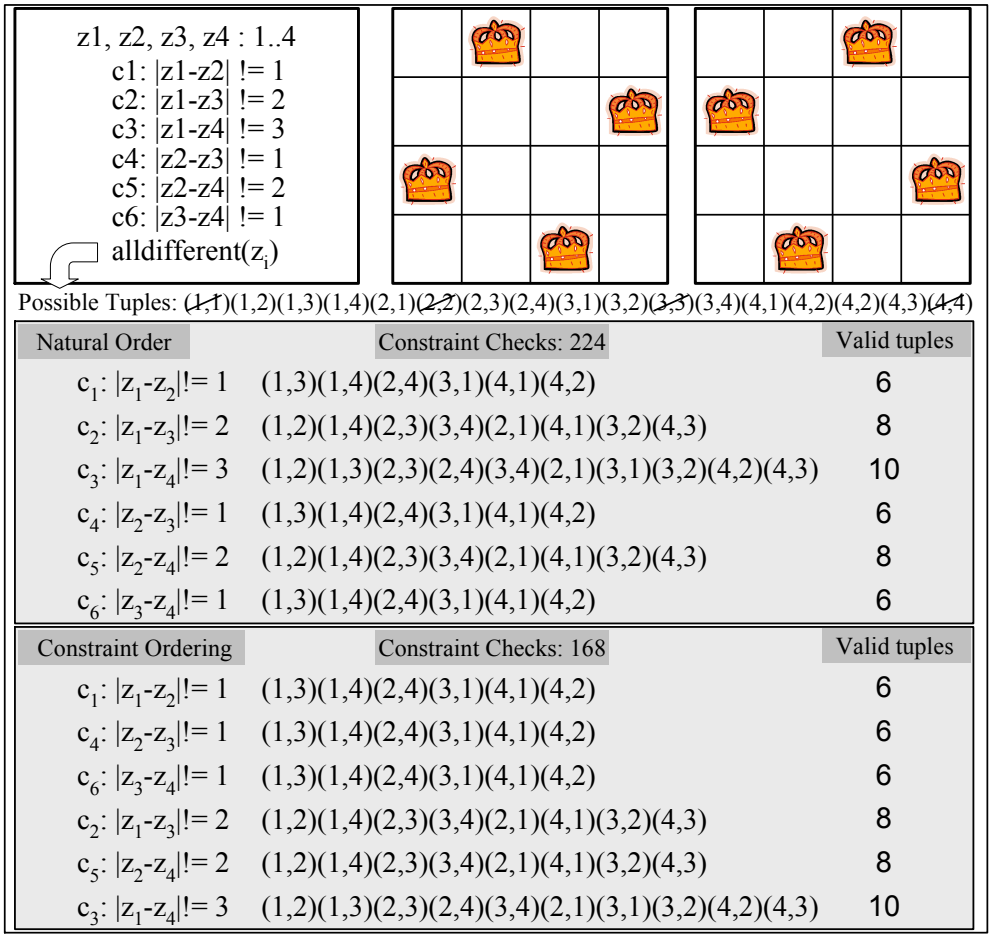


Figura 2.17: Problema de las 4-reinas con restricciones originales y restricciones ordenadas con COH.

Ejemplo: Cuando COH es aplicado al problema de las 4-Reinas, presentado en la Figura 2.17, con un nivel de confianza del 75 %, obtenemos el siguiente valor de s:

$$s(4, 4, 0.25) := \lceil \frac{4^4}{1+4^4 \cdot 0.25^2} \rceil = 16$$

La Figura 2.17 muestra el problema de 4-reinas con dos órdenes distintos de restricciones: orden natural -donde se efectúan 224 chequeos- y orden obtenido al aplicar COH -donde se efectúan 168 chequeos-.

2.6. Normalización de un CSP no-normalizado

La única manera de convertir un CSP no-normalizado en uno normalizado es a través de la representación extensional de las restricciones.

Es bien conocido que una restricción puede ser representada intensionalmente (por una expresión) o extensionalmente (por el conjunto de tuplas válidas o no válidas). La gran mayoría de restricciones que presentan los problemas reales son modeladas intensionalmente. Algunas veces estas restricciones son representadas extensionalmente para ser procesadas por los resolutores de CSP, técnicas de filtrado, etc. Sin embargo, esta es una tarea muy dura, particularmente en dominios grandes, o imposible, en dominios continuos. En realidad, desde el punto de vista matemático, las representaciones extensionales e intensionales son iguales. Sin embargo, desde la perspectiva computacional, la representación intensional es mucho más compacta y expresiva que la representación extensional.

Por ejemplo, para $D_1 = D_2 = \{0, 1, 2\}$, una restricción R_{12} expresada intensionalmente como $R_{12} : X_1 < X_2$ puede ser extensionalmente definida por:

- tuplas permitidas (válidas): $R_{12} = \{(0, 1), (0, 2), (1, 2)\}$ o
- tuplas no permitidas: $R_{12} = \{(0, 0), (1, 0), (1, 1), (2, 0), (2, 1), (2, 2)\}$

En CSPs con dominios grandes, como por ejemplo, $D_1 = D_2 = \{1, 2, ..., 1000\}$, una restricción R_{12} con la representación extensional genera un millón de tuplas $(D_1 x D_2)$, las cuales serán etiquetadas como tuplas permitidas o no permitidas. Esta tarea tiene una alta complejidad temporal y espacial.

Una vez que todas las restricciones del CSP no-normalizado han sido convertidas a una representación extensional (conjunto de tuplas permitidas o no permitidas), las restricciones que involucran las mismas variables deben ser agrupadas con el fin de seleccionar la intersección de todas las tuplas. Por ejemplo, si existen dos restricciones que involucran a las variables X_i y X_j (R_{ij}, R'_{ij}), y S_{ij}, S'_{ij} son los conjuntos de tuplas permitidas, respectivamente, entonces $C_{ij} = \{(a, b)|(a, b) \in S_{ij} \wedge (a, b) \in S'_{ij}\}$

Para el ejemplo de Figura 2.18 las restricciones no normalizadas son (R_{12} y R'_{12}). El Producto Cartesiano de los dominios de las variables es:

$D_1 \text{x} D_2 = \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)\}$.

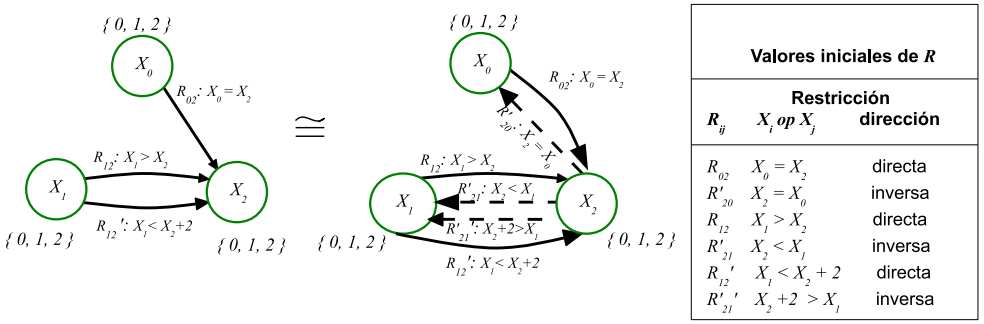


Figura 2.18: Ejemplo de CSP binario no-normalizado.

Las tuplas permitidas de la restricción R_{12} son $S_{12} = \{(1,0),(2,0),(2,1)\}$, mientras que las tuplas permitidas para las restricción R'_{12} son:

$S'_{12} = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,1),(2,2)\}$;

De modo que $C_{12} = S_{12} \cap S'_{12} = \{(1,0),(2,1)\}$. Así, mediante la aplicación de la arco-consistencia, los dominios quedan podados a $D_1 = \{1,2\}$ y $D_2 = \{0,1\}$. Aunque esta técnica de conversión es bastante simple, su implementación consume mucho tiempo y requiere la generación de estructuras de datos que consumen memoria.

En conclusión, el coste de convertir un CSP no-normalizado en uno normalizado es una tarea prohibitiva en dominios grandes. Las principales investigaciones realizadas en relación con las técnicas de filtrado para la satisfacción de restricciones ha centrado la atención en CSPs normalizados, o en representaciones de restricciones extensionales. Sin embargo, en problemas de la vida real son representados usualmente como CSPs no-normalizados, con restricciones representadas intensionalmente. Por lo que el desarrollo de técnicas de filtrado, para manejar este tipo de problemas, es necesario.

2.7. Conclusiones

En este capítulo, se ha efectuado una revisión sobre los principales conceptos y técnicas relacionadas con los problemas de satisfacción de restricciones tales como: técnicas de búsqueda, técnicas de consistencia y heurísticas de ordenación. Se ha realizado especial énfasis en las técnicas de arco-consistencia, presentando para cada una de ellas, sus algoritmos, su funcionamiento y sus limitaciones. En esta área se

puede evidenciar el interés mostrado por la comunidad científica en la investigación sobre problemas binarios normalizados.

Capítulo 3

Algoritmos de arco-consistencia

En problemas de satisfacción de restricciones la arco-consistencia garantiza que cada valor en el dominio de cada variable esté soportado por un valor de la otra variable que participe en la restricción. En este capítulo, presentamos cinco algoritmos para realizar el proceso de arco-consistencia. Cada uno de estos algoritmos reformulan al algoritmo que tiene como base y evita las deficiencias detectadas que han sido señaladas en el Capítulo 2. También mejoran su desempeño o permiten trabajar con restricciones no-normalizadas. La Tabla 3.1 muestra características generales de los algoritmos propuestos, tales como: el algoritmo base que reformulan, su granularidad (forma de propagar las podas), el tipo de restricciones que pueden manejar (normalizadas y no-normalizadas) y el nivel de consistencia que alcanzan.

Algoritmo Propuesto	Algoritmo Base	granularidad	Para restricciones			Nivel de Consistencia
			binarias	normalizadas	no-normalizadas	
AC3-OP	AC3	gruesa	✓	✓		arco-consistencia
AC3-NN	AC3	gruesa	✓	✓	✓	arco-consistencia
AC4-OP	AC4	fina	✓	✓		arco-consistencia
AC4-OPNN	AC4-OP	fina	✓	✓	✓	arco-consistencia
AC2001-OP	AC2001/3.1	gruesa	✓	✓		arco-consistencia

Tabla 3.1: Algoritmos de filtrado propuestos.

En las siguientes secciones de este capítulo se explicará en detalle cada uno de los algoritmos propuestos. A continuación explicaremos brevemente la funcionalidad de cada uno:

- AC3-OP [9, 11] mejora el algoritmo de arco-consistencia AC3, en restricciones de orden, logrando reducir el número de propagaciones hasta en un 50 % y con ello el tiempo y el número de chequeos de restricciones.

- AC3-NN [12] es una reformulación del algoritmo AC3 para procesar restricciones no-normalizadas y realizar correctamente la propagación de arcos que comparten las mismas variables.

- AC4-OP [10] poda el mismo espacio de búsqueda que AC4, pero mejora su eficiencia tanto en la inicialización (que la realiza bidireccionalmente, ya que sólo chequea las restricciones binarias normalizadas en un sentido) como en la propagación (sólo con variables que sean soporte de otras). AC4-OP es capaz de reducir el número de chequeos de restricciones en un 50 % y el número de propagaciones en un 15 %.

- AC4-OPNN [12, 6] poda el mismo espacio de búsqueda que AC4 y AC4-OP, pero mejora su eficiencia tanto en la inicialización que la realiza bidireccionalmente (sólo chequea las restricciones binarias no-normalizadas en un sentido) como en la propagación (sólo con variables que sean soporte de otras). Al igual que AC4-OP, AC4-OPNN es capaz de reducir el número de chequeo de restricciones en un 50 % y el número de propagaciones en un 15 % en CSPs no-normalizados.

- AC2001-OP [15] mejora el algoritmo óptimo de arco-consistencia AC2001/3.1, logrando reducir en restricciones de orden el número de propagaciones hasta en un 71 % y con ello el tiempo en un 47 %.

3.1. AC3-OP

El algoritmo AC3-OP (ver Algoritmo 17) es un algoritmo de granularidad gruesa que evita las ineficiencias de propagación que sufre el algoritmo AC3, que señalamos en el capítulo anterior. AC3-OP puede realizar menos chequeos de restricciones y ser más eficiente que AC3. Para ello, etiquetamos cada restricción en R entre las variables X_i y X_j (arco) de la siguiente forma:

- $(R_{ij} \equiv (R_{ij}, 1))$ restricción directa (la que viene dada en el problema original).

- $(R_{ji} \equiv (R_{ij}, 2))$ restricción inversa.

Algoritmo 17: Procedimiento AC3-OP

Datos: Un CSP, $P = \langle X, D, R \rangle$

Resultado: **verdadero** y P' (que es arco-consistente) o **falso** y P' (el cual es inconsistente porque se ha vaciado un dominio)

```
1  principio
2      para cada arco R_ij ∈ R hacer
3          Añadir (QS, (R_ij, 1))
4          Añadir (QS, (R_ij, 2))
5      mientras QS ≠ ∅ hacer
6          Seleccione y elimine (R_ij, s) de la pila QS | s = 1 ∨ s = 2
7          si ReviseAC3((R_ij, s)) = verdadero entonces
8              si D_i ≠ ∅ y D_j ≠ ∅ entonces
9                  si ((opR_ij =' <' y s = 1) o (opR_ij ≠' <')) entonces
10                     Añadir (QS, (R_ki, 1)) con k ≠ i, k ≠ j
11                 sino
12                     Añadir (QS, (R_ki, 1)) con k ≠ i, k ≠ j, opR_ki ≠' <'
13                 Añadir (QS, (R_ki, 2)) con k ≠ i, k ≠ j, opR_ki ≠' <'
14             sino
15                 retornar falso /*dominio vacío*/
16     retornar verdadero
17 fin
```

La estructura de datos QS, que utiliza AC3-OP para almacenar las restricciones, inicialmente actúa como una cola y posteriormente como una pila. De esta manera, en los pasos 2 al 4 del Algoritmo 17, QS es inicializado insertando las tuplas $(R_{ij}, 1)$ y $(R_{ij}, 2)$ como una cola. A partir de este punto, QS será procesado como una pila.

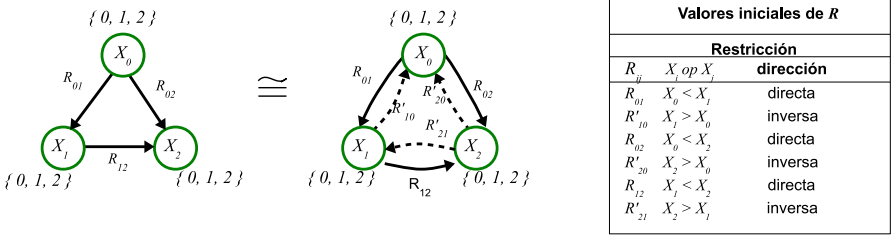
AC3-OP utiliza el mismo procedimiento *Revise* que AC3. En el paso 7, si $s = 1$ entonces el procedimiento *Revise* recibe la restricción directa R_{ij}. Si $s = 2$, entonces recibe la restricción inversa R_{ji}.

AC3-OP añadirá nuestras restricciones al tope de QS en los siguientes casos:

- si el operador de la restricción es $<$ y $s = 1$, añade restricciones directas (paso 10);
- si el operador de la restricción es distinto a $<$, añade restricciones directas e

inversas (pasos 12 y 13).

Las restricciones añadidas serán nuevamente seleccionadas en el paso 6, es decir, antes que el resto de las restricciones previamente almacenadas en QS. Así también, la inclusión de restricciones sólo es efectuada si dichas restricciones no están presentes en QS.



Valores iniciales de R		
Restricción		
R_{ij}	X_i op X_j	dirección
R_{01}	$X_0 < X_1$	directa
R'_{10}	$X_1 > X_0$	inversa
R_{02}	$X_0 < X_2$	directa
R'_{20}	$X_2 > X_0$	inversa
R_{12}	$X_1 < X_2$	directa
R'_{21}	$X_2 > X_1$	inversa

Figura 3.1: Ejemplo de CSP binario normalizado.

Utilizando el mismo ejemplo que en el Capítulo 2 (ver Figura 3.1), AC3 realizó 9 iteraciones para alcanzar la arco-consistencia (en ambas modalidades cola y pila), sin embargo AC3-OP sólo realizó 7 iteraciones en el mismo problema.

La Tabla 3.2 muestra las iteraciones realizadas por AC3-OP en la resolución del ejemplo de la Figura 3.1. Así, en la iteración 1, AC3-OP evalúa la restricción directa $R_{01} : X_0 < X_1$, es podado el valor 2 de D_{X_0} y no se añaden propagaciones, ya que las restricciones que podrían ser propagadas están actualmente en QS esperando ser evaluadas. En la iteración 2, AC3-OP evalúa la restricción inversa $R_{01} : X_1 > X_0$, es podado el valor 0 de D_{X_1} y por ser una restricción inversa con operador '>', AC3-OP no realiza ningún tipo de propagación (AC3 [83] si hubiera realizado propagación). En la iteración 3, AC3-OP evalúa la restricción directa $R_{02} : X_0 < X_2$ y al no efectuar ninguna poda, tampoco realiza propagaciones. La iteraciones 4 y 7 son similares a la iteración 2 (con diferentes variables), ya que AC3-OP evalúa restricciones inversas, hace poda y no realiza propagaciones (por ser restricciones inversas con operador '>'). En la iteración 5, AC3-OP al evaluar la restricción directa $R_{12} : X_1 < X_2$, poda el valor 2 de D_{X_1} y en la propagación apila en QS la restricción R_{01}, que será evaluada en la siguiente iteración (6), antes que la restricción R_{12} (que se evalúa en la iteración 7).

Tabla 3.2: Iteraciones realizadas por AC3-OP para el ejemplo de la Figura 3.1.

Iter	(R_{ij}, s)	Restricción X_i op X_j	Poda X_i	Add Restr.	Comentarios
1	$(R_{01}, 1)$	$X_0 < X_1$	$X_0 = 2$		
2	$(R_{01}, 2)$	$X_1 > X_0$	$X_1 = 0$		s=2 no adic.
3	$(R_{02}, 1)$	$X_0 < X_2$	*None*		
4	$(R_{02}, 2)$	$X_2 > X_0$	$X_2 = 0$		s=2 no adic.
5	$(R_{12}, 1)$	$X_1 < X_2$	$X_1 = 2$	$(R_{01}, 1)$	
6	$(R_{01}, 1)$	$X_0 < X_1$	$X_0 = 1$		
7	$(R_{12}, 2)$	$X_2 > X_1$	$X_2 = 2$		s=2 no adic.

La Figura 3.2 compara AC3 (almacenado en cola y pila) con AC3-OP para el ejemplo mostrado en la Figura 3.1. Podemos visualizar que AC3 (cola), AC3 (pila) y AC3-OP efectúan la misma poda pero difieren en el número de iteraciones y en las iteraciones donde son realizadas las podas. Cabe destacar que las iteraciones adicionales que realiza AC3 incluyen también chequeos de restricciones innecesarios, puesto que no efectúan ninguna poda.

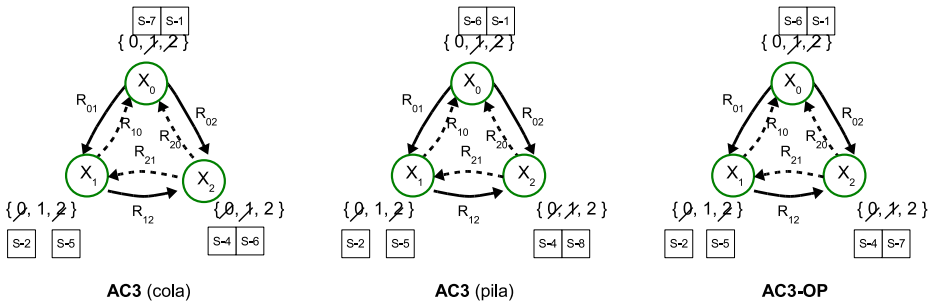


Figura 3.2: Iteraciones (S-i) y valores podados utilizando AC3 (cola), AC3 (pila) y AC3-OP para el ejemplo de la Figura 3.1.

3.2. AC3-NN

El algoritmo AC3-NN es un algoritmo de granularidad gruesa que alcanza la arco-consistencia en problemas binarios no-normalizados (ver Algoritmo 18). Originalmente el algoritmo AC3 fue diseñado para grafos dirigidos (únicamente procesa R_{ij} y no procesa R_{ji}). Subsecuentes referencias a AC3 encontradas en la literatura [24, 28, 41] suponen que las restricciones del CSP hay que evaluarlas en ambas direcciones: restricción directa R_{ij} y restricción inversa R_{ji}, y que las restricciones son binarias y normalizadas.

Algoritmo 18: Procedimiento AC3-NN

Datos: A CSP, $P = \langle X, D, R \rangle$
Resultado: **verdadero** y P' (el cual es arco-consistente) o **falso** yd P' (el cual es inconsistente porque se ha vaciado un dominio)

```
1  principio
2      C = ∅
3      n = 0
4      para cada restricción R_ij ∈ R hacer
5          Nuevo (C_n)
6          Asignar C_n.id = n y C_n.R = R_ij
7          Añadir (C, (C_n))
8          n++
9          Nuevo (C'_n)
10         Asignar C'_n.id = n y C'_n.R = R'_ji
11         Añadir (C, (C'_n))
12         n++
13     para cada restricción C_n ∈ C hacer
14         Añadir (Q, (C_n))
15     mientras Q ≠ ∅ hacer
16         Seleccione y borre C_n de la cola Q
17         R_ij = C_n.R
18         si Revise(R_ij) = vardadero entonces
19             si D_i ≠ ∅ entonces
20                 Añadir (Q, (C_k)) con (C_k.id ≠ C_n.id) y (C_k.R = R_ki); ∀ C_k ∈ C
21             sino
22                 retornar falso /*dominio vacío*/
23     retornar verdadero
24 fin
```

Sin embargo, un CSP puede ser no-normalizado, esto es, existe más de una restricción entre dos variables ($R_{ij}, R'_{ij}, R''_{ij}$, etc). Para poder diferenciar las restricciones que involucran el mismo par de variables (restricciones no-normalizadas), AC3-NN almacena cada restricción R_{ij} en una tupla $\langle C.id, C.R \rangle$, donde $C.id$ almacena un identificador (número entero) y $C.R$ almacena la restricción en sí. Para

distinguir las restricciones directas de las restricciones inversas utilizamos la siguiente notación: C para restricción directa y C' para la restricción inversa (el Algoritmo 18, pasos 2-12, muestra la creación de C y C' para AC3-NN). De esta forma, AC3-NN puede realizar correctamente la propagación de restricciones no-normalizadas que deben ser re-evaluadas, y también hace posible el utilizar el mismo procedimiento *Revise* de AC3 [83] sin realizar ningún cambio, ya que este procedimiento seguirá evaluando arcos.

El cuerpo principal de AC3-NN es un ciclo (ver Algoritmo 18, pasos 15 -22) que selecciona y revisa las estructuras C y C' almacenadas en una cola Q, hasta que no ocurran cambios (Q se ha vaciado) o se vacíe el dominio de una variable. El primer caso asegura que todos los valores de los dominios son arco-consistentes con todas las restricciones y el segundo caso indica que el problema es inconsistente.

Para evitar muchas llamadas inútiles al procedimiento *Revise*, AC3-NN almacena en una cola Q, estructuras C_k para los cuales no tiene garantía de que D_i sea arco-consistente con R_{ij}. Además, antes de añadir C_k en la cola Q, aseguramos que el dominio de la variable X_i podada no se haya vaciado. (Ver Algoritmo 18, pasos 19 y 20).

En relación al ejemplo mostrado en la Figura 2.18, la Tabla 3.3 muestra las iteraciones que realiza AC3-NN. Así, en la iteración 4, AC3-NN evalúa la restricción inversa $C'_{21} : X2 < X1$. AC3-NN encuentra soporte en D_{X_1} para los valores 0 y 1 $\in$ D_{X_2}, pero no consigue soporte en D_{X_1} para el valor 2 $\in D_{X_2}$, por lo que dicho valor es podado y realizarse su propagación. La propagación hace que se añada a la cola Q la restricción C_{02} , que será evaluada en la iteración 7. Trabajando de esta forma, AC3-NN realiza: 3 podas de valores, 32 chequeos de restricciones y 1 propagación en Q, para alcanzar la arco-consistencia en un problema no-normalizado.

3.3. AC4-OP

El Algoritmo AC4-OP (ver Algoritmo 19), es un algoritmo de grano fino, que permite alcanzar la arco-consistencia en problemas binarios normalizados. AC4-OP encuentra todos los soportes de cada valor de una variable, realizando el chequeo de restricciones bidireccionalmente.

AC4-OP sigue la estructura de dos fases de AC4 [89], pero evita sus ineficiencias,

Tabla 3.3: Iteraciones (Iter) realizadas por AC3-NN para el ejemplo mostrado en la Figura 2.18.

Iter	Nodo (C_{id}, C_R)	val a	val b	Satisf.	Poda	Adic. Q
1	$(C_0, C_{01}) : X_0 = X_2$	0	1	si		
		1	0	no		
			1	si		
		2	0	no		
			1	no		
			2	si		
2	$(C_1, C'_{10}) : X_2 = X_0$	0	0	si		
		1	0	no		
			1	si		
		2	0	no		
			1	no		
			2	si		
3	$(C_3, C_{12}) : X_1 > X_2$	0	0	no		
			1	no		
			2	no	$X_1 = 0$	
		1	0	si		
		2	0	si		
4	$(C_3, C'_{21}) : X_2 < X_1$	0	1	si		
		1	1	no		
			2	si		
		2	1	no		
			2	no	$X_2 = 2$	(C_0, C_{02})
5	$(C_4, C_{12}) : X_1 < X_2 + 2$	1	1	si		
		2	1	no		
			2	si		
6	$(C_5, C'_{21}) : X_2 + 2 > X_1$	0	1	si		
		1	1	si		
7	$(C_0, C_{02}) : X_0 = X_2$	1	0	si		
		1	0	no		
			1	si		
		2	0	no		
			1	no	$X_0 = 2$	

las cuales son:

1. Cada vez que el procedimiento *InitializationAC4* evalúa una restricción, almacena en M información sobre la presencia o no de cada valor en la variable y sus soportes (cuáles son y cuántos son) en S y $Counter$, respectivamente;

2. la restricción directa R_{ij} y la restricción inversa R'_{ji} comparten las mismas variables (X_i y X_j); y

3. por definición de simetría de la restricción, el soporte es bidireccional.

Debido a los puntos 1 y 2, hay ineficiencias en el Algoritmo AC4 porque los valores de M, S y $Counter$ pueden ser actualizados para $\langle X_j, b\rangle$ cuando la restricción directa es evaluada. En este punto, únicamente se pierden los valores de X_j que pueden ser podados (si hay alguno) porque el ciclo interno es ejecutado en varias ocasiones (es decir, una vez para cada valor $a \in D_i$, véase Algoritmo 20 pasos 9 al 14). Como indica el punto 3, si hay soportes ($total > 0$), entonces son ejecutados los pasos 23-24 del Algoritmo 20. No obstante, AC4 únicamente actualiza la variable X_i e ignora el hecho de que puede también actualizar la variable X_j.

AC4-OP (comienza con la fase de inicialización, ver paso 2 del Algoritmo 19) en la cual realiza la actualización bidireccional de la variable X_j de la siguiente forma: inicializa su $Counter[X_j, b, X_i]$ (ver Algoritmo 20, paso 5); computa el nuevo soporte en $Counter[X_j, b, X_i]$ (paso 13), y poda los valores inconsistentes de D_j en los pasos 25 al 33. Adicionalmente, AC4-OP únicamente propaga aquellas tuplas soportadas por otras tuplas, (véanse los pasos 20-21 y 31-32 del Algoritmo 20, y los pasos 13-14 del Algoritmo 19). Así, AC4-OP evita realizar propagaciones ineficientes en Q, y con ello también evita los chequeos ineficientes.

Algoritmo 19: Procedimiento AC4-OP

```
   Datos: Un CSP, P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es arco-consistente) o  falso y P' (el cual es
              arco-inconsistente)
 1 principio
 2     initial ← InitilizeAC4OP(P)
 3     si initial = verdadero entonces
 4         mientras Q ≠ φ hacer
 5             Seleccione y elimine (X_j, b) de la cola Q
 6             para cada (X_i, a) ∈ S[X_j, b] hacer
 7                 Counter[X_i, a, X_j] ← Counter[X_i, a, X_j] − 1
 8                 si Counter[X_i, a, X_j] = 0 ∧ M[X_i, a] = 1 entonces
 9                     elimine a de D_i
10                     si D_i = φ entonces
11                         retornar falso
12                     sino
13                         si S[X_i, a] ≠ {} entonces
14                             Q ← Q ∪ (X_i, a)
15                         M[X_i, a] ← 0
16         retornar verdadero
17     sino
18         retornar falso
19 fin
```

La Figura 3.3 muestra las estructuras una vez efectuada la evaluación de los valores de X_0. Esta evaluación es completada en el paso 24 del Algoritmo 20. Las líneas continuas en la Figura 3.3 reflejan el estado de las estructuras S, M y $Counter$ para la variable X_0 (hasta ahora, las mismas estructuras de AC4). Las estructuras S, M y $Counter$ para la variable X_1 son mostradas en líneas punteadas. Estas estructuras de X_1 fueron actualizadas por AC4-OP mientras buscaba los soportes para X_0. Así, AC4-OP realiza las actualizaciones de ambas variables cuando procesa la restricción R_{01}, pero primero debe completar la asignación de todos los valores de X_0 para determinar la poda de los valores de X_1.

Algoritmo 20: Procedimiento InitializeAC4OP

Datos: $P = \langle X, D, R\rangle$ /*R contiene únicamente restricciones directas*/
Resultado: initial=**verdadero** y P', Q, S, M, $Counter$ o initial=**falso** y P' (el cual es arco-inconsistente).

```
1  principio
2      Q ← {}
3      S[X_j, b] ←← {}  /* ∀X_j ∈ X ∧ ∀b ∈ D_j */
4      M[X_i, a] ← 1  /* ∀X_i ∈ X ∧ ∀a ∈ D_i */
5      Counter[X_i, a, X_j] ← 0  /* ∀X_i, X_j ∈ X ∧ ∀a ∈ D_i */
6      para cada arco R_ij ∈ R hacer
7          para cada a ∈ D_i hacer
8              total ← 0
9              para cada b ∈ D_j hacer
10                 si (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ R_ij entonces
11                     total ← total + 1
12                     Añadir (S[X_j, b], (X_i, a))
13                     Counter[X_j, b, X_i] ← Counter[X_j, b, X_i] + 1
14                     Añadir (S[X_i, a], (X_j, b))
15             si total = 0 entonces
16                 Eliminar a de D_i
17                 si D_i = φ entonces
18                     retornar initial = falso
19                 sino
20                     si S[X_i, a] ≠ {} entonces
21                         Q ← Q ∪ (X_i, a)
22                 M[X_i, a] = 0
23             sino
24                 Counter[X_i, a, X_j] = total
25         para cada b ∈ D_j hacer
26             si Counter[X_j, b, X_i] = 0 entonces
27                 Eliminar b de D_j
28                 si D_j = φ entonces
29                     retornar initial = falso
30                 sino
31                     si S[X_j, b] ≠ {} entonces
32                         Q ← Q ∪ (X_j, b)
33                 M[X_j, b] = 0
34     retornar initial = verdadero y Q, M, S, Counter
35 fin
```

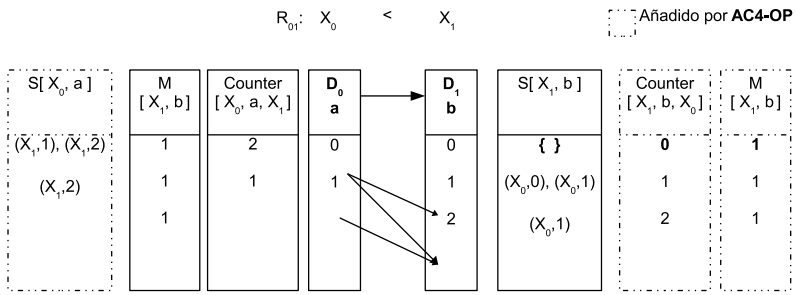


Figura 3.3: Una iteración realizada por InitializationAC4-OP hasta el paso 24 del Algoritmo 20, para el ejemplo de Figura 3.1.

Debido a que todos los *Counters* de X_0 tienen valor mayor que 0, no hay podas en X_0. Sin embargo, para X_1, $Counter[X_1, 0, X_0] = 0$ indica que el valor $0 \in D_1$ no tiene soporte en X_0. Adicionalmente, $M[X_1, 0] = 1$ indica que el valor 0 está aún incluido en el dominio de X_1 (véase la Figura 3.3). Como ambas condiciones son verdaderas (véase pasos 25-26, del Algoritmo 20), el valor 0 es podado de D_1. Si se vaciara el dominio, el problema sería inconsistente; en caso contrario S y M son actualizadas (ver pasos 31-33, del Algoritmo 20). Se puede observar que la tupla $\langle X_1, 0\rangle$ no es añadida a la cola Q para propagar los efectos de la poda, debido a que $S[X_1, 0] = \{\}$ (no soporta a ninguna variable). La Figura 3.4 muestra el proceso completado.

Realizando el proceso de inicialización descrito anteriormente, AC4-OP no requiere revisar la restricción inversa (R'_{10}). Esto es debido a que ha evaluado la variable X_1 en la revisión de la variable X_0 con la restricción R_{01}. Así, AC4-OP reduce en un 50 % el promedio de chequeo de restricciones.

Las Tablas 3.4 y 3.5 muestran las inicializaciones y propagaciones realizadas por AC4-OP para el ejemplo de Figura 3.1. Las estructuras S, M, y $Counter$ para AC4-OP son las mismas que para AC4, pero AC4-OP sólo realiza 19 chequeos de restricciones para el mismo problema (47 % menos que AC4) y realiza 3 propagaciones (50 % menos que AC4) mientras obtiene la misma poda (6 valores para este problema).

Finalmente, la Figura 3.5 muestra una comparación entre AC4 y AC4-OP, donde

Tabla 3.4: Iteraciones efectuadas por el procedimiento *InitializeAC4OP* para el ejemplo de la Figura 3.1.

Iter 1. $R_{02} : X_0 < X_2$						
D_0 a	D_2 b	Counter $[X_i, a, X_j]$	Poda X_i	Counter $[X_j, b, X_i]$	Poda X_j	Añade Q
0	0	0		0		
	1	1		1		
	2	2		1		
1	0	0		0		
	1	0		1		
	2	1		2		
2	0	0		0		
	1	0		1		
	2	0	$X_0 = 2$	2		
	0			0	$X_2 = 0$	
	1			1		
	2			2		

Iter 2. $R_{01} : X_0 < X_1$						
D_0 a	D_1 b	Counter $[X_i, a, X_j]$	Podas X_i	Counter $[X_j, b, X_i]$	Poda X_j	Añade Q
0	0	0		0		
	1	1		1		
	2	2		1		
1	0	0		0		
	1	0		1		
	2	1		2		
	0			0	$X_1 = 0$	
	1			1		
	2			2		

Iter 3. $R_{12} : X_1 < X_2$						
D_1 a	D_2 b	Counter $[X_i, a, X_j]$	Poda X_i	Counter $[X_j, b, X_i]$	Poda X_j	Añade Q
1	1	0		0		
	2	1		1		
2	1	0		0		
	2	0	$X_1 = 2$	1		$\langle X_1, 2\rangle$
	1			0	$X_2 = 1$	$\langle X_2, 1\rangle$
	2			1		

Tabla 3.5: Iteraciones efectuadas por AC4-OP (únicamente propagación) para el ejemplo de la Figura 3.1.

Iter	tupla $\langle X_j, b\rangle$	Soportes $S[X_j, b]$	Counter $[X_i, a, X_j]$	Poda X_i	Añade Q $\langle X_i, a\rangle$
1	$\langle X_1, 2\rangle$	$\langle X_0, 0\rangle$	1		
		$\langle X_0, 1\rangle$	0	$\langle X_0, 1\rangle$	$\langle X_0, 1\rangle$
2	$\langle X_2, 1\rangle$	$\langle X_0, 0\rangle$	1		
3	$\langle X_0, 1\rangle$	$\langle X_2, 2\rangle$	1		
		$\langle X_1, 2\rangle$	1		

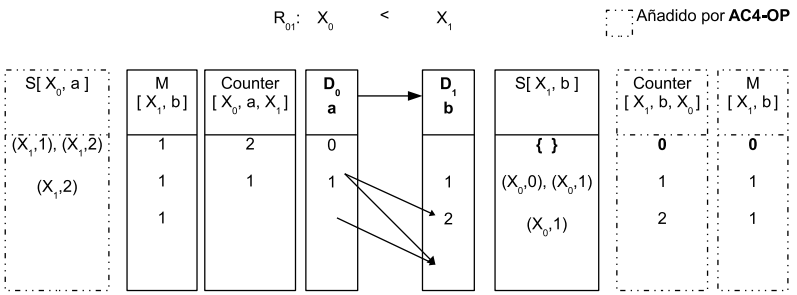


Figura 3.4: Una iteración realizada por InitializationAC4-OP hasta el paso 33 del Algoritmo 20 para el ejemplo de Figura 3.1. El valor $0 \in D_1$ ha sido podado y ha cambiado $M[X_1, 0] = 0$.

puede observarse que ambos alcanzan la misma poda, pero difieren en el número de iteraciones y el lugar donde realizan las podas.

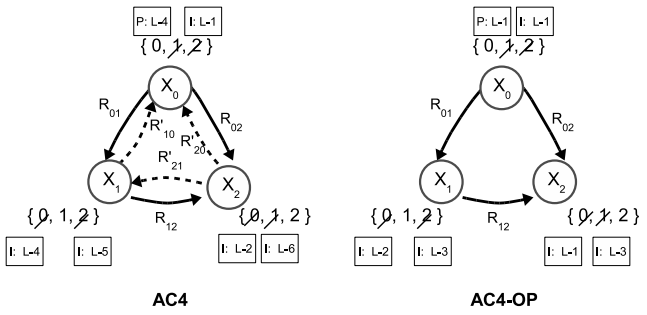


Figura 3.5: Iteración (L-i) y valores podados utilizando AC4 y AC4-OP, para el ejemplo en la Figura 3.1. **P** significa fase de propagación e **I** significa fase de inicialización.

3.4. AC4-OPNN

El algoritmo AC4-OPNN es un algoritmo de grano-fino, que realiza poda de valores inconsistentes en problemas binarios no-normalizados. El nivel máximo de consistencia que puede alcanzar es arco-consistencia, pero ello no ocurre en todos los

casos. AC4-OPNN utiliza estructuras similares a AC4 y AC4-OP, a las que añade un vector llamado $suppInv$. Este vector es utilizado para almacenar los soportes inversos de cada valor de una variable, los cuales no pueden ser almacenados en $Counter$ debido a que el mismo par de variables están presentes en más de una restricción. El tamaño de $suppInv$ es la talla del dominio más grande ($maxD$). Así, una vez que la revisión de los valores $a \in D_i$ son actualizados ($Counter[X_i, a, X_j]$), podemos podar los valores $b \in D_j$ cuyo $suppInv[b] = 0$ (si los hubiera). De esta forma, no se requiere evaluar la restricción inversa R'_{ji}.

El número de soportes encontrados por cada valor $a \in D_i$ son inicialmente almacenados en una variable local llamada *total*. Si no hay un valor de soporte b en D_j para el valor a, (es decir, $total = 0$), entonces el valor a es eliminado de D_i y ambas, la matriz M y la cola Q, son actualizadas. En caso contrario, la matriz $Counter$ es incrementada con el valor almacenado en *total*. En este punto del Algoritmo, la matriz $Counter$ almacena el número de soportes de cada valor para CSPs normalizados y no-normalizados.

Una vez que se ha completado la fase de Inicialización, comienza la fase de Propagación. Esta fase consiste en *propagar* las consecuencias de eliminar (podar o borrar) valores en los dominios de las variables. Así, las tuplas $\langle variable, valor \rangle$ almacenadas en Q son seleccionadas y revisadas hasta que no ocurran cambios (Q se ha quedado sin tuplas) o hasta que se haya vaciado el dominio de una variable. En el primer caso, el algoritmo asegura que la mayor parte de los valores del dominio son consistentes, mientras que en el segundo caso, el algoritmo indica que el problema no tiene solución.

Así, el Algoritmo 21 inicializa las estructuras $Counter[X_j, b, X_i]$ y $suppInv[b]$ en los pasos 5 y 6, respectivamente; en el paso 14, el algoritmo añade un nuevo soporte en $Counter[X_j, b, X_i]$; en el paso 15, cuenta un nuevo soporte en $suppInv[b]$; y en los pasos 27 al 37, poda los valores inconsistentes de D_j.

Debido al hecho de que el mismo par de variables X_i y X_j pueden estar presentes en más de una restricción R_{ij} (CSPs no-normalizados), los contadores $Counter$ pueden guardar valores de restricciones evaluadas anteriormente las cuales comparten las mismas variables. Por esta razón, las podas son efectuadas en función a los contadores de cada variable en cada restricción. El contador de soportes de la variable X_i (*total*) es inicializado a 0 para cada valor $a \in D_i$ (véase Algoritmo 21,

Algoritmo 21: Procedimiento InitializeAC4OPNN

```
Datos: P = ⟨X, D, R⟩  /*R contiene únicamente restricciones directas*/
Resultado: initial=verdadero y P', Q, S, M, Counter o initial=falso y P' (el cual es
           inconsistente).
1  principio
2      Q ← {}
3      S[X_j, b] ← {}  /* ∀X_j ∈ X ∧ ∀b ∈ D_j */
4      M[X_i, a] ← 1  /* ∀X_i ∈ X ∧ ∀a ∈ D_i */
5      Counter[X_i, a, X_j] ← 0  /* ∀X_i, X_j ∈ X ∧ ∀a ∈ D_i */
6      suppInv[b] ← 0  /* ∀b ∈ [1, maxD] */
7      para cada restricción R_ij ∈ R hacer
8          para cada a ∈ D_i hacer
9              total ← 0
10             para cada b ∈ D_j hacer
11                 si (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ R_ij entonces
12                     total ← total + 1
13                     Añadir (S[X_j, b], ⟨X_i, a⟩)
14                     Counter[X_j, b, X_i] ← Counter[X_j, b, X_i] + 1
15                     suppInv[b] ← suppInv[b] + 1
16                     Añadir (S[X_i, a], ⟨X_j, b⟩)
17             si total = 0 entonces
18                 Eliminar a de D_i
19                 si D_i = φ entonces
20                     retornar initial = falso
21                 sino
22                     si S[X_i, a] ≠ {} entonces
23                         Q ← Q ∪ ⟨X_i, a⟩
24                 M[X_i, a] ← 0
25             sino
26                 Counter[X_i, a, X_j] ← Counter[X_i, a, X_j] + total
27         para cada b ∈ D_j hacer
28             si suppInv[b] = 0 entonces
29                 Eliminar b de D_j
30                 si D_j = φ entonces
31                     retornar initial = falso
32                 sino
33                     si S[X_j, b] ≠ {} entonces
34                         Q ← Q ∪ ⟨X_j, b⟩
35                 M[X_j, b] ← 0
36             sino
37                 suppInv[b] ← 0
38     retornar initial = verdadero y Q, M, S, Counter
39 fin
```

Algoritmo 22: Procedimiento AC4-OPNN

```
   Datos: CSP, P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es arco-consistente) o  falso y P' (el cual es inconsistente)
 1 principio
 2     initial= InitilizeAC4OPNN(P)
 3     si initial = verdadero entonces
 4         mientras Q ≠ φ hacer
 5             Seccione y elimine ⟨X_j, b⟩ de la cola Q
 6             para cada ⟨X_i, a⟩ ∈ S[X_j, b] hacer
 7                 Counter[X_i, a, X_j] ← Counter[X_i, a, X_j] − 1
 8                 si Counter[X_i, a, X_j] ← 0 ∧ M[X_i, a] = 1 entonces
 9                     Elimine a de D_i
10                     si D_i = φ entonces
11                         retornar falso
12                     sino
13                         si S[X_i, a] ≠ {} entonces
14                             Q ← Q ∪ ⟨X_i, a⟩
15                     M[X_i, a] ← 0
16         retornar verdadero
17     sino
18         retornar falso
19 fin
```

paso 9). Sin embargo, el contador de soportes de variable X_j (soportes inversos) debe ser dividido en dos contadores distintos: $Counter[X_j, b, X_i]$ y $suppInv[b]$. El vector $suppInv$ almacena el número de soportes de cada valor de X_j. Este vector es inicializado a cero (véase Algoritmo 21, paso 6). Cuando el valor $b \in D_j$ soporta el valor $a \in D_i$, se incrementará $suppInv[b]$ (ver Algoritmo 21, paso 15). Durante el ciclo formado por los pasos 8-26, el vector es actualizado, para posteriormente ser analizado en el paso 28. Una vez completado el procesamiento de todos los valores de D_i, si un valor b de D_j no tiene ningún soporte ($suppInv[b] = 0$), entonces este valor es podado de D_j y se propaga su poda. Si $suppInv[b] > 0$, entonces b está soportado y $suppInv[b]$ es inicializado a 0 para poder usar posteriormente este vector en la siguiente restricción (ver Algoritmo 21, pasos 27 a 37).

Al igual que AC4-OP, AC4-OPNN sólo propaga las tuplas que sean soporte de otras tuplas, evitando así propagaciones y chequeos ineficientes de tuplas en Q. Aunque AC4-OPNN también es válido para CSPs normalizados, es ineficiente, ya que requiere una estructura adicional $suppInv$ que consume memoria, por lo que en ese caso se recomienda utilizar AC4-OP.

Para el ejemplo mostrado en la Figura 3.6, R sólo almacena las restricciones

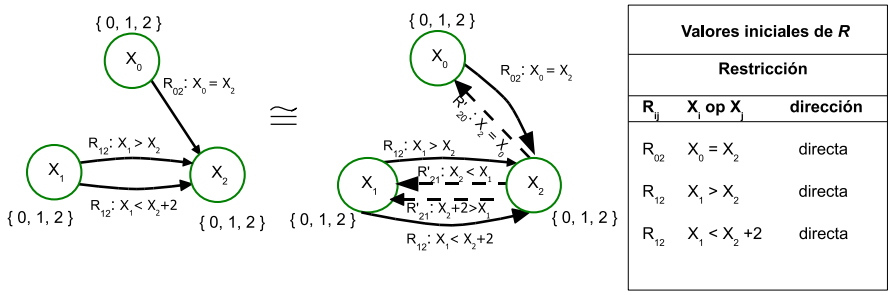


Figura 3.6: Ejemplo de CSP binario no-normalizado con R almacenando sólo restricciones directas.

directas del problema que se muestran en la Tabla. Las Tablas 3.9 y 3.10 muestran las inicializaciones y propagaciones efectuadas por AC4-OPNN para dicho ejemplo.

Las estructuras M, S y $Counter$ para AC4-OPNN son las mismas que para AC4 (ver Tablas 3.7, 3.63.8). Los valores de los dominios de las variables están reflejados en la matriz M, la cual es inicializada con todos los valores en 1. Así, por ejemplo, en la iteración 2 (ver Tabla 3.9) de la fase de Inicialización son podados los valores 0 y 2 del dominio de las variables X_1 y X_2, respectivamente. Por lo que aparecen en la Tabla 3.6, columna Ini con el valor 0. En la iteración 1 de la fase de propagación es podado el valor 2 de X_0, lo cual se ve reflejado en la columna prop de la matriz M. Dado que AC4-OPNN realiza tres podas, M solo refleja tres variables con valor 0. En cuanto a las estructuras $Counter$ y S, puede observarse que, en la iteración 1 de la fase de inicialización (ver Tabla 3.9) cuando se chequea el valor $0 \in D_0$ éste consigue un soporte con el valor $0 \in D_2$, lo cual queda reflejado en la Tabla 3.7 fila 1, $S[X_0, 0]$, en la columna Ini, contiene la tupla $\langle X_2, 0\rangle$ y también en la Tabla 3.8 en la fila 1, columna Ini, contiene el valor de 1.

Así, AC4 realiza 3 podas de valores, 41 chequeos de restricciones (Cc) y 3 propagaciones (Np) en Q, para alcanzar la arco-consistencia. Por otro lado, AC4-OPNN realiza 3 podas, 22 chequeos de restricciones para el mismo problema (47 % menos que AC4). Además, sólo realiza 2 propagaciones (33 % menos que AC4).

Tabla 3.6: Cambios en la matriz M después de ejecutarse las fases de inicialización (Ini) y propagación (Prop) por AC4-OPNN ,para el ejemplo de la Figura 3.6.

$M[var, val]$	**Comienzo**	Ini	Prop
$M[X_0, 0]$	1		
$M[X_0, 1]$	1		
$M[X_0, 2]$	1		0
$M[X_1, 0]$	1	0	
$M[X_1, 1]$	1		
$M[X_1, 2]$	1		
$M[X_2, 0]$	1		
$M[X_2, 1]$	1		
$M[X_2, 2]$	1	0	

Tabla 3.7: Cambios en la matriz S después de ejecutarse las fases de inicialización (Ini) y propagación (Prop) por AC4-OPNN, para el ejemplo de la Figura 3.6.

$S[X_j, b]$	**Comienzo**	Ini
$S[X_0, 0]$	{}	$\{\langle X_2, 0\rangle\}$
$S[X_0, 1]$	{}	$\{\langle X_2, 1\rangle\}$
$S[X_0, 2]$	{}	$\{\langle X_2, 2\rangle\}$
$S[X_1, 0]$	{}	
$S[X_1, 1]$	{}	$\{\langle X_2, 0\rangle, \langle X_2, 0\rangle, \langle X_2, 1\rangle\}$
$S[X_1, 2]$	{}	$\{\langle X_2, 0\rangle, \langle X_2, 1\rangle, \langle X_2, 1\rangle\}$
$S[X_2, 0]$	{}	$\{\langle X_0, 0\rangle, \langle X_1, 1\rangle, \langle X_1, 2\rangle, \langle X_1, 1\rangle\}$
$S[X_2, 1]$	{}	$\{\langle X_0, 1\rangle, \langle X_1, 2\rangle, \langle X_1, 1\rangle, \langle X_1, 2\rangle\}$
$S[X_2, 2]$	{}	$\{\langle X_0, 2\rangle\}$

Tabla 3.8: Cambios en la matriz *Counter* después de ejecutarse las fases de inicialización (Ini) y propagación (Prop) por AC4-OPNN, para el ejemplo de la Figura 3.6.

$Counter[X_i, a, X_j]$	**Comienzo**	Ini	Prop		$Counter[X_i, a, X_j]$	**Comienzo**	Ini	Prop
$Counter[X_0, 0, X_2]$	0	1			$Counter[X_1, 0, X_2]$	0	0	
$Counter[X_0, 1, X_2]$	0	1			$Counter[X_1, 1, X_2]$	0	3	
$Counter[X_0, 2, X_2]$	0	1	0		$Counter[X_1, 2, X_2]$	0	3	
$Counter[X_0, 0, X_1]$	0				$Counter[X_2, 0, X_0]$	0	1	
$Counter[X_0, 1, X_1]$	0				$Counter[X_2, 1, X_0]$	0	1	
$Counter[X_0, 2, X_1]$	0				$Counter[X_2, 2, X_0]$	0	1	0
$Counter[X_1, 0, X_0]$	0				$Counter[X_2, 0, X_1]$	0	3	
$Counter[X_1, 1, X_0]$	0				$Counter[X_2, 1, X_1]$	0	3	
$Counter[X_1, 2, X_0]$	0				$Counter[X_2, 2, X_1]$	0	0	

Tabla 3.9: Iteraciones realizadas por InitializeAC4-OPNN para el ejemplo de la Figura 3.6.

Iter 1. $R_{02} : X_0 = X_2$

D_0 a	D_2 b	total	suppInv $[j]$	Counter $[X_i, a, X_j]$	Poda X_i	Counter $[X_j, b, X_i]$	Poda X_j	Añade Q
0	0	1	1			1		
	1	1	0			0		
	2	1	0	1		0		
1	0	0	1			1		
	1	1	1			1		
	2	1	0	1		0		
2	0	0	1			1		
	1	0	1			1		
	2	1	1	1		1		
	0		0			1		
	1		0			1		
	2		0			1		

Iter 2. $R_{12} : X_1 > X_2$

D_0 a	D_2 b	total	suppInv $[j]$	Counter $[X_i, a, X_j]$	Poda X_i	Counter $[X_j, b, X_i]$	Poda X_j	Añade Q
0	0	0	0			0		
	1	0	0			0		
	2	0	0	0	$X_1 = 0$	0		
1	0	1	1			1		
	1	1	0			0		
	2	1	0	1		0		
2	0	1	2			2		
	1	2	1			1		
	2	2	0	2		0		
	0		0			2		
	1		0			1		
	2		0			0	$X_2 = 2$	$\langle X_2, 2\rangle$

Iter 3. $R_{12} : X_1 < X_2 + 2$

D_0 a	D_2 b	total	suppInv $[j]$	Counter $[X_i, a, X_j]$	Poda X_i	Counter $[X_j, b, X_i]$	Poda X_j	Añade Q
1	0	1	1			3		
	1	2	1	3		2		
2	0	0	1			3		
	1	1	2	3		3		
	0		0			3		
	1		0			3		

Tabla 3.10: Iteraciones realizadas por AC4-OPNN (sólo propagación) para el ejemplo de la Figura 3.6.

Iter	tupla $\langle X_j, b\rangle$	Soportes $S[X_j, b]$	Counter $[X_i, a, X_j]$	Poda X_i	Añade Q $\langle X_i, a\rangle$
1	$\langle X_2, 2\rangle$	$\langle X_0, 2\rangle$	0	$X_0 = 2$	$\langle X_0, 2\rangle$
2	$\langle X_0, 2\rangle$	$\langle X_2, 2\rangle$	0		

3.5. AC2001-OP

El algoritmo AC2001/3.1 [33], presentado en el Capítulo 2, es eficiente en la realización del proceso de arco-consistencia, porque evita los re-chequeos de restricciones al verificar si el valor de soporte b previamente encontrado para $\langle X_i, a\rangle$, es aún válido (i.e., $b \in D_j$) y además realiza la búsqueda de un nuevo soporte a partir del último valor de soporte encontrado. Sin embargo, AC2001/3.1 es ineficiente en cuanto a que no actualiza los soportes encontrados bidireccionalmente (esta ineficiencia fue resuelta en el algoritmo AC3.3 [76]); y como en otras técnicas de arco-consistencia, AC2001/3.1 siempre propaga cuando realiza podas, sin considerar si la propagación será efectiva (permite hacer más podas) o inefectiva (ninguna poda es realizada), como sucede con la propagación de algunas restricciones aritméticas inversas $(R_{ij}, 2) \in R$.

Por lo anteriormente expuesto, proponemos el algoritmo AC2001-OP, el cual consiste en una reformulación del algoritmo AC2001/3.1 en su estrategia de propagación. El algoritmo AC2001-OP utiliza procedimientos Initialize2001 y Revise2001 similares a los de AC2001/3.1, pero con algunos cambios para evitar las ineficiencias detectadas. Dichos cambios son:

- Cada restricción R_{ij} entre las variables X_i y X_j es etiquetada como una tupla (R_{ij}, s), donde $s = 1$ si es una restricción directa $(R_{ij} \equiv (R_{ij}, 1))$ y $s = 2$ si es una restricción inversa $(R_{ji} \equiv (R_{ij}, 2))$.

- Q es cambiado por QS, donde las tuplas (R_{ij}, s) que se almacenan inicialmente en QS son encoladas (modo FIFO de las colas) y posteriormente las tuplas que se añaden en las propagaciones son apiladas (comportamiento LIFO de las pilas). Así, QS almacena tuplas $(R_{ij}, s)\,|s \in \{1, 2\}$ que deben ser chequeadas.

- Presenta un nuevo esquema de propagación en función del operador de la restricción y del valor s (ver Algoritmo 23, pasos 9 a 14).

Añadiendo los cambios mencionados, el algoritmo AC2001-OP alcanza la arco-consistencia, con menos propagaciones, menos chequeos y en menor cantidad de tiempo que AC2001/3.1.

La estructura QS, utilizada por AC2001-OP para almacenar las restricciones que deben ser evaluadas, actúa en primer lugar como una cola (FIFO) y luego como

Algoritmo 23: Procedimiento AC2001-OP

```
   Datos: Un CSP, P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es arco-consistente) o falso y P' (el cual es inconsistente
              porque se ha tornado vacío uno de sus dominios)
 1 principio
 2     Initialize2001(P, Last)
 3     para cada arco R_ij ∈ R hacer
 4         Añadir (QS, (R_ij, 1))
 5         Añadir (QS, (R_ij, 2))
 6     mientras QS ≠ φ hacer
 7         Seleccione y elimine (R_ij, s) de la pila QS donde s = 1 ∨ s = 2
 8         si Revise2001(R_ij, s) = verdadero entonces
 9             si D_i ≠ φ y D_j ≠ φ entonces
10                 si ((opR_ij =' <' y s = 1) o (opR_ij ≠' <')) entonces
11                     Añadir (QS, (R_ki, 1)) con k ≠ i, k ≠ j
12                 sino
13                     Añadir (QS, (R_ki, 1)) con k ≠ i, k ≠ j, opR_ki ≠' <'
14                 Añadir (QS, (R_ki, 2)) con k ≠ i, k ≠ j, opR_ki ≠' <'
15             sino
16                 retornar falso /*dominio vacío*/
17     retornar verdadero
18 fin
```

una pila (LIFO). QS es inicializada (pasos 3 a 5, del Algoritmo 23) insertando las tuplas $(R_{ij}, 1)$ y $(R_{ij}, 2)$ en forma FIFO. Posteriormente, QS se comportará como una pila (LIFO), lo cual le permite propagar la poda de valores, eliminando en fases tempranas valores que se han vuelto inconsistentes por la eliminación de su soporte. Con ello se evita que dichos valores sean evaluados posteriormente. Si $s = 1$ (paso 8 de AC2001-OP), entonces el Procedimiento *Revise* recibe la restricción directa R_{ij}. Si $s = 2$ en el paso 8, entonces recibe la restricción inversa R_{ji}.

AC2001-OP añadirá nuevas restricciones al tope de QS en tres casos diferentes:

- si el operador de la restricción es '$<$', $s = 1$ y las posibles restricciones a ser añadidas no estén previamente guardadas en QS (paso 11);
- en el resto de operadores $\{\leq, =, \neq\}$, si las posibles restricciones a ser añadidas en QS no están ya guardadas en QS (paso 11 o paso 13);
- si el operador de restricción es '$<$', $s = 2$ y el operador de las posibles restricciones a ser añadida es distinto a '$<$' y que dicha restricción no esté actualmente guardada en QS (paso 14).

Las restricciones añadidas serán seleccionadas nuevamente en el paso 7, es decir,

antes que el resto de las restricciones almacenadas previamente en el la estructura QS. A la estructura QS, sólo se agregan aquellas restricciones que no están incluidas a la misma.

3.6. Conclusiones

Trabajando sobre los algoritmos de arco-consistencia existentes, en este capítulo hemos presentado una serie de algoritmos que alcanzan la arco-consistencia en problemas binarios normalizados (algoritmos: AC3-OP, AC4-OP y AC2001-OP) y problemas binarios no-normalizados (algoritmos: AC3-NN, y AC4-OPNN), logrando mejorar la eficiencia en cuanto a uno o más de los siguientes factores: número de podas, chequeos de restricciones, número de propagaciones y tiempo de cómputo. En el Capítulo 6 se evaluará el comportamiento de estas propuestas frente a a algoritmos bien conocidos como AC3, AC4, AC2001/3.1, etc.

Capítulo 4

Algoritmos de 2-Consistencia

En los capítulos anteriores se observa que la mayoría de algoritmos que realizan la arco-consistencia asumen que los problemas son binarios y normalizados. Si relajamos la asunción de que las restricciones están normalizadas y trabajamos con problemas con restricciones no-normalizadas, los algoritmos de arco-consistencia no son capaces de realizar la misma cantidad de podas que las técnicas de 2-consistencia, a menos que se realice un proceso de normalización.

La 2-consistencia garantiza que cualquier instanciación de un valor a una variable pueda ser consistentemente extendida a una segunda variable. La arco-consistencia sólo verifica que el soporte sea válido en una restricción, mientras que la 2-consistencia verifica que el soporte sea válido en todas las restricciones que comparten las mismas variables. Así, en CSPs binarios, la 2-consistencia puede ser más restrictiva y realizar más poda que la arco-consistencia.

En este capítulo, presentamos seis algoritmos que alcanzan la 2-consistencia. La Tabla 4.1 proporciona una visión general de los algoritmos propuestos, en cuanto al algoritmo base que les sirve de marco, su granularidad (fina o gruesa), el tipo de restricciones que pueden manejar (normalizadas y no-normalizadas) y el nivel de consistencia que alcanzan.

Algoritmo Propuesto	Algoritmo Base	granularidad	Para restricciones binarias		Nivel de Consistencia
			normalizadas	no-normalizadas	
2-C3	AC3	gruesa	✓	✓	2-consistencia
2-C3OP	2-C3,AC7	gruesa	✓	✓	2-consistencia
2-C3OPL	2-C3,AC2001/3.1, AC3rm	gruesa	✓	✓	2-consistencia
2-C4	AC4, 2-C3, 2-C3OP	fina	✓	✓	2-consistencia
2-C6	AC6, 2-C3	fina	✓	✓	2-consistencia
AC3NH(*)	AC3	gruesa	✓	✓	2-consistencia
(*) AC3NH logra alcanzar la 2-consistencia gracias al proceso de normalización que realiza.					

Tabla 4.1: Algoritmos de filtrado propuestos.

Seguidamente, explicaremos brevemente la funcionalidad de cada algoritmo desarrollado. En las siguientes secciones serán explicados con mayor detalle.

- 2-C3 [7] es un algoritmo donde las restricciones no-normalizadas son agrupadas en conjuntos C_{ij}, por lo que cada instanciación $\langle a, b\rangle$ será válida si satisface todas las restricciones presentes en el conjunto C_{ij}, lo cual permite alcanzar la 2-consistencia; lográndose una poda superior en un 50 % con respecto a los algoritmos de arco-consistencia, en problemas aleatorios no-normalizados con restricciones con operadores $op \in \{<, \leq, =, \neq, >, \geq\}$ y manteniendo tiempos de cómputo similares a los de AC3.

- 2-C3OP [8, 13] al igual que el algoritmo 2-C3, agrupa las restricciones en conjuntos C_{ij} para procesarlos bidireccionalmente posteriormente. Alcanza así la 2-consistencia. Su poder de poda puede ser superior en un 50 % con respecto a los algoritmos de arco-consistencia, con mejor tiempo de cómputo que el resto de los algoritmos evaluados, menor número de chequeos de restricciones que AC3 y 2-C3, y menor número de propagaciones que AC3.

- 2-C3OPL [16, 17] es un algoritmo que utiliza la estructura C_{ij} de 2-C3, la matriz *Last* de AC2001/3.1, aplicando bidireccionalidad y como AC3rm, almacenando el último soporte encontrado. Optimiza el proceso de 2-consistencia de los algoritmos 2-C3 y 2-C3OP, debido a que 2-C3OPL tiene el mejor tiempo de cómputo que el resto de los algoritmos evaluados, menor número de chequeos de restricciones y menor número de propagaciones. 2-C3OPL es el algoritmo de pre-proceso utilizado en las técnicas de búsqueda BLS y SchTrains que se explicarán en el próximo capítulo.

- 2-C4 [19] es un algoritmo que utiliza las estructuras de AC4 y los conjuntos de restricciones C_{ij} de 2-C3. Procesa bidireccionalmente las restricciones en C_{ij} para alcanzar la 2-consistencia. 2-C4 busca todos los soportes para cada una de las variables, lo cual, combinado con un algoritmo de búsqueda, permite realizar MAC en $O(1)$.

- 2-C6 es una reformulación del algoritmo AC6. Permite alcanzar la 2-consistencia y realizar las propagaciones en grano-fino. Similar al algoritmo 2-C3, el algoritmo 2-C6 agrupa las restricciones en conjuntos C_{ij}, sobre los cuales realiza la

posterior revisión. 2-C6, al igual que 2-C4, trabaja en 2 fases: Inicialización y Propagación, pero a diferencia de éste, limita la búsqueda a un único soporte.

- AC3NH [14] es un algoritmo de normalización híbrido que alcanza la 2-consistencia. AC3NH es capaz de detectar y normalizar las restricciones no-normalizadas, sin modificar las restricciones normalizadas del problema original. Posteriormente utiliza el algoritmo AC3 para realizar la consistencia. En la evaluación empírica se comprobó que AC3N realiza la misma cantidad de podas que 2-consistencia, pero el coste de normalización no es trivial en lo que se refiere al número de chequeos de restricciones y tiempo de cómputo.

4.1. 2-C3

El algoritmo 2-C3 alcanza la 2-consistencia en CSP binarios y no-normalizados. Este algoritmo es una reformulación del algoritmo AC3. El cuerpo principal de 2-C3 (ver Algoritmo 24) consiste en un ciclo que selecciona y revisa (ver Algoritmo 25) conjuntos de restricciones almacenadas en la cola Q, hasta que no ocurran cambios (Q está vacía), o hasta que el dominio de una variable se vacíe. El primer caso garantiza que todos los valores de los dominios sean 2-consistentes. El segundo caso indica que el problema no tiene solución.

El procedimiento *Revise* de 2-C3 es muy similar al procedimiento *Revise* de AC3. La única diferencia consiste en que la instanciación $\langle X_i = a, X_j = b\rangle$ debe ser chequeada con el conjunto de restricciones C_{ij}, en vez de hacerlo con una única restricción R_{ij}. El conjunto de restricciones en C_{ij} puede también ser ordenado evitando así chequeos innecesarios. Si ordenamos este conjunto, desde las restricciones más restrictivas a las menos restrictivas, el chequeo de las restricciones podrá encontrar antes las inconsistencias y evitar así que se lleven a cabo chequeos innecesarios [107].

En el ejemplo 4.1, tenemos un CSP no-normalizado, compuesto de tres variables y tres restricciones. En la cola Q, se almacenan inicialmente, los siguientes conjuntos de restricciones: $Q = \{C_{02}, C'_{20}, C_{12}, C'_{21}\}$.

La Tabla 4.2 muestra cómo el procedimiento 2-C3 evalúa cada restricción. En las iteraciones 1 y 2, los conjuntos C_{02} y C'_{02} tienen una sola restricción: R_{02} y R'_{02},

Algoritmo 24: Procedimiento 2-C3

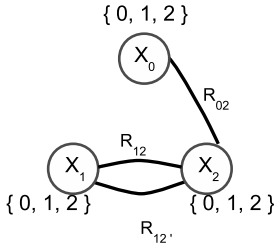
Datos: A CSP, $P = \langle X, D, R \rangle$
Resultado: **verdadero** y P' (el cual es 2-consistente) **o** **falso** y P' (el cual es inconsistente porque se ha vaciado un dominio)

```
1  principio
2      para cada i, j hacer
3          C_ij = ∅
4      para cada arco R_ij ∈ R hacer
5          C_ij ← C_ij ∪ R_ij
6      para cada conjunto C_ij hacer
7          Q ← Q ∪ {C_ij, C'_ji}
8      mientras Q ≠ φ hacer
9          seleccione y elimine C_ij de la cola Q
10         si Revise2C3(C_ij) = verdadero entonces
11             si D_i ≠ φ entonces
12                 Q ← Q ∪ {C_ki | k ≠ i, k ≠ j}
13             sino
14                 retornar falso /*dominio vacío*/
15     retornar verdadero
16 fin
```

Algoritmo 25: Procedimiento Revise2C3

Datos: Un $CSPP'$ definido por dos variables $X = (X_i, X_j)$, dominios D_i y D_j, y el conjunto de restricciones C_{ij}.
Resultado: D_i, de forma tal que la variable X_i es 2-consistente en relación con la variable X_j y la variable Booleana *change*

```
1  principio
2      change ← falso
3      para cada a ∈ D_i hacer
4          si ∄b ∈ D_j de tal manera que (X_i = a, X_j = b) ∈ C_ij entonces
5              Eliminar a de D_i
6              change ← verdadero
7      retornar change
8  fin
```



$C_{02} = \{ R_{02} \}$
$C_{12} = \{ R_{12}\ ,\ R_{12'} \}$
$C'_{20} = \{ R'_{02} \}$
$C'_{21} = \{ R'_{21}\ ,\ R'_{21'} \}$

R_i	Restricción X_i op X_j	Arco (X_i, X_j)
R_{02}	$X_0 = X_2$	(X_0, X_2)
R'_{20}	$X_2 = X_0$	(X_2, X_0)
R_{12}	$X_1 <= X_2$	(X_1, X_2)
R'_{21}	$X_2 => X_1$	(X_2, X_1)
$R_{12'}$	$X_1 <> X_2$	(X_1, X_2)
$R'_{21'}$	$X_2 <> X_1$	(X_2, X_1)

Figura 4.1: Ejemplo de CSP binario no-normalizado. Los algoritmos de arco-consistencia no realizan ninguna poda, a menos que se realize un proceso de normalización sobre las restricciones del problema.

respectivamente. De esta forma, los chequeos para estos conjuntos son equivalentes a los realizados por AC3, pero en las iteraciones 3 y 4, cada conjunto tiene dos restricciones. Luego, 2-C3 chequea que las instanciaciones satisfagan a todas las restricciones almacenadas en el conjunto. Así, en la iteración 3, es procesado el bloque de restricciones C_{12} para verificar si el valor 0 de D_1 tiene un soporte con el valor 0 de D_2. Esto es cierto para la restricción R_{12} pero es falso para la restricción (R'_{12}), ya que esta última restricción no es satisfecha. Por esta razón, un nuevo valor en D_2 (valor 1) es solicitado, y de esta forma $X_1 = 0$ y $X_2 = 1$ satisfacen ambas restricciones. En las líneas punteadas añadimos el bloque de restricciones C_{02} que debe ser re-evaluado. Este bloque fue añadido a la cola Q (Ver Tabla 4.2, iteración 4).

4.2. 2-C3OP

El algoritmo 2-C3OP [8] es un algoritmo de grano grueso que alcanza la 2-consistencia en CSPs binarios no-normalizados. 2-C3OP trabaja con conjuntos de restricciones como 2-C3 [7] pero sólo requiere manejar la mitad de restricciones. Utiliza dos procedimientos: *Revise2C3OP* y *AddQ*. 2-C3OP realiza los chequeos bi-direccionalmente (como AC7) y realiza inferencia para evitar chequeos innecesarios. Sin embargo, la forma de hacerlo difiere de AC7, puesto que requiere las siguientes estructuras, que son compartidas por todas las restricciones:

Tabla 4.2: Iteraciones (Iter) realizadas por 2-C3 para el ejemplo mostrado en la Figura 4.1.

Iter	Conjunto C_{ij}	val a	val b	R_{ij}	hold	cambia	Poda X_i	Add Q
1	C_{02}	0	0	R_{02}	si			
		1	0	R_{02}	no			
			1	R_{02}	si			
		2	0	R_{02}	no			
			1	R_{02}	no			
			2	R_{02}	si	falso		
2	C'_{20}	0	0	R'_{20}	si			
		1	0	R'_{20}	no			
			1	R'_{20}	si			
		2	0	R'_{20}	no			
			1	R'_{20}	no			
			2	R'_{20}	si	falso		
3	C_{12}	0	0	R_{12}	si			
				$R_{12'}$	no			
			1	R_{12}	si			
				$R_{12'}$	si			
		1	0	R_{12}	no			
			1	R_{12}	si			
				$R_{12'}$	no			
			2	R_{12}	si			
				$R_{12'}$	si			
		2	0	R_{12}	no			
			1	R_{12}	no			
			2	R_{12}	si			
				$R_{12'}$	no	verdadero	$\langle X_1, 2\rangle$	
4	C'_{21}	0	0	R'_{21}	si			
				$R'_{21'}$	no			
			1	R'_{21}	no	verdadero	$\langle X_2, 0\rangle$	
		1	0	R'_{21}	si			
				$R'_{21'}$	si			
		2	0	R'_{21}	si			
				$R'_{21'}$	si			C_{02}
5	C_{02}	0	1	R_{02}	no	verdadero	$\langle X_2, 0\rangle$	
			2	R_{02}	no			
		1	1	R_{02}	si			
		2	1	R_{02}	no			
			2	R_{02}	si			

- *suppInv*: es un vector cuyo tamaño es la talla máxima del dominio del problema ($maxD$). Almacena el valor de X_i que soporta al valor de X_j.

- *minSupp*: es una variable entera que almacena el primer valor $b \in D_j$ que soporta a un valor $a \in D_i$. Utilizando *minSupp* se realiza inferencia que evita chequeos innecesarios, porque los valores $b \in D_j < minSupp$ son podados sin realizar chequeos, cuando es evaluado $suppInv$.

- t: es una variable entera y puede tomar los siguientes valores $t = \{1, 2, 3\}$. Este valor es utilizado para guiar al procedimiento *Revise*. Según su valor, se verificará o no el conjunto de restricciones C_{ij} (en forma directa o inversa) y se aplicará o no bidireccionalidad en la búsqueda.

Algoritmo 26: Procedimiento 2-C3OP

```
   Datos: A CSP, P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es 2-consistente) o falso y P' (el cual es inconsistente
              porque se ha vaciado un dominio )
 1 principio
 2     para cada i, j hacer
 3         C_ij = ∅
 4     para cada arco R_ij ∈ R hacer
 5         C_ij ← C_ij ∪ R_ij
 6     para cada conjunto C_ij hacer
 7         Q ← Q ∪ {(C_ij, t) : t = 1}
 8     para cada d ∈ D_max hacer
 9         suppInv[d] ← 0
10     mientras Q ≠ ∅ hacer
11         Seleccione y elimine (C_ij, t) de la cola Q con t = {1, 2, 3}
12         change ← Revise2C3OP((C_ij, t))
13         si change > 0 entonces
14             si change ≥ 1 ∧ change ≤ 3 entonces
15                 Q ← Q ∪ AddQ(change, (C_ij, t))
16             sino
17                 retornar falso /*dominio vacío*/
18     retornar verdadero
19 fin
```

Inicialmente, el procedimiento 2-C3OP almacena la tuplas $(C_{ij}, t) : t = 1$ en la cola Q. A continuación, se ejecuta un ciclo donde se selecciona y revisa las tuplas almacenadas en Q hasta que no ocurran cambios (Q está vacío), o hasta que se vacíe el dominio de una variable. El primer caso asegura que cada valor es 2-consistente y el segundo caso indica que el problema es inconsistente.

Algoritmo 27: Procedimiento Revise2C3OP

Datos: Un CSP P' definido por dos variables $X = (X_i, X_j)$, los dominios D_i y D_j, la tupla (C_{ij}, t) y el vector $suppInv$.
Resultado: D_i, de tal forma que X_i es 2-consistente en relación a X_j y D_j,y que X_j es 2-consistente en relación a X_i y la variable entera $change$

```
1  principio
2      change_i ← 0 y change_j ← 0
3      minSupp ← dummy_value
4      para cada a ∈ D_i hacer
5          si ∄b ∈ D_j de tal forma que (X_i = a, X_j = b) ∈ (C_ij, t) entonces
6              eliminar a de D_i
7              change_i ← 1
8          sino
9              suppInv[b] ← 1
10             si minSupp = dummy_value entonces
11                 minSupp ← b
12     si ([(t = 2 ∨ t = 3) ∧ change_i = 1] ∨ t = 1) entonces
13         para cada b ∈ D_j hacer
14             si b < minSupp entonces
15                 Eliminar b de D_j
16                 change_j ← 2
17             sino
18                 si suppInv[b] > 0 entonces
19                     suppInv[b] ← 0
20                 sino
21                     si ∄a ∈ D_i de tal forma que (X_i = a, X_j = b) ∈ (C_ij, t) entonces
22                         Eliminar b de D_j
23                         change_j ← 2
24     change ← change_i + change_j
25     retornar change
26 fin
```

El procedimiento Revise2C3OP (ver Algoritmo 27) requiere dos variables internas denominadas $change_i$ y $change_j$. Estas variables son inicializadas a cero y son usadas para recordar los dominios podados. Por ejemplo: si D_i fue podado, entonces $change_i = 1$ y si D_j fue podado, entonces $change_j = 2$. Sin embargo, si ambos dominios D_i y D_j fueron podados, entonces $change = 3$ (porque $change = change_i + change_j$).

En el proceso de revisión, durante el ciclo de pasos del 4 al 11 del Algoritmo 27, cada valor de D_i es chequeado[1]. Si el valor $b \in D_j$ soporta al valor $a \in D_i$, entonces $suppInv[b] = a$. Por la simetría de las restricciones (el soporte es bidireccional). Adicionalmente, el primer valor $b \in D_j$ (el cual soporta un valor en D_i) es guardado en la variable $minSupp$.

La segunda parte del Algoritmo 27 es llevada a cabo en función de los valores t y $change_i$. Si $t = 2$ o $t = 3$, y $change_i = 0$, entonces C_{ij} no necesita ser chequeda ya que la restricción no ha tenido poda en la iteración previa. Sin embargo, si $t = 1$, entonces C_{ij} requiere una evaluación completa bidireccional. Si $t = 2$ o $t = 3$, y $change_i = 1$, entonces C_{ij} requiere también una evaluación completa bidireccional. En ambos casos, el procesamiento de $suppInv$ puede ser realizado de tres formas diferentes: 1) valores $b \in D_j : b < minSupp$ son eliminados sin realizar ningún chequeo. Adicionalmente, la variable $change_j$ es actualizada a $change_j = 2$ para indicar que ha ocurrido un cambio en el segundo ciclo. 2) Los valores de b con $suppInv[b] > 0$ no serán chequeados ya que ellos están soportados en el dominio D_i; y el valor de $suppInv[b]$ es actualizado a 0, para uso posterior de este vector. 3) Los valores con $b > minSupp$ y $suppInv[b] = 0$ tendrán que ser chequeados hasta que les sea encontrado un soporte $a \in D_i$ o sean eliminados (por no estar soportados). En este último caso, la variable $change_j$ es asignada con el valor 2 para indicar que ha ocurrido un cambio en el dominio.

El procedimiento $AddQ$ (ver Algoritmo 31) añade las tuplas que deben ser evaluadas nuevamente. La adición de tuplas dependerá de las tuplas que están presentes en Q y del valor de la variable $change$. Dependiendo del valor t, generado por $AddQ$, el procedimiento Revise23C3OP guiará la búsqueda. Si $t = 1$ la búsqueda es realizada bidireccionalmente (directa e inversa); si $t = 2$ la búsqueda es inversa y será directa si y solo si es efectuada alguna poda; finalmente, si $t = 3$ la búsqueda es directa y

[1] si t=2 el operador inverso es usado

se hará inversa si y solo si se ha efectuado una poda.

Algoritmo 28: Procedimiento AddQ

```
   Datos: Variable entera change, tupla (C_ij, t) y cola Q.
   Resultado: Cola Q actualizada
 1 principio
 2     si ((change = 1 ∧ t = 2) ∨ (change = 2 ∧ t = 1)) entonces
 3         c ← j
 4     si ((change = 2 ∧ t = 2) ∨ (change = 1 ∧ t = 1)) entonces
 5         c ← i
 6     si (change = 3) entonces
 7         c ← j y o ← j
 8     para cada C_kl con C_kl ≠ C_ij hacer
 9         si change = 3 entonces
10             Q ← Q ∪ {(C_kl, 1)|(c = l ∨ o = k) ∨ (c = k ∧ o ≠ l) ∨ (c ≠ k ∧ o = l)}
11         sino
12             si ((t = 2 ∧ c = k) ∨ (t = 1 ∧ c = l)) entonces
13                 Q ← Q ∪ {(C_kl, 3)|[(C_kl, 1) ∨ (C_kl, 3)] ∉ Q}
14             si ((t = 2 ∧ c = l) ∨ (t = 1 ∧ c = k)) entonces
15                 Q ← Q ∪ {(C_kl, 2)|[(C_kl, 1) ∨ (C_kl, 2) ∨ (C_kl, 3)] ∉ Q}
16 fin
```

Considerando el ejemplo de la Figura 4.1, 2-C3OP inicialmente almacena en Q los conjuntos de restricciones directas C_{02} y C_{12}. 2-C3OP realiza: 3 podas, 24 chequeos de restricciones (Cc) (ver Tabla 4.4), y 1 propagación (Np) en Q (ver Tabla 4.3) para alcanzar la 2-consistencia. 2-C3 realiza: 3 podas, 37 chequeos de restricciones (Cc) y 1 propagación (Np) en Q, para alcanzar la 2-consistencia. Pese a realizar 29 chequeos de restricciones, AC3 no realiza ninguna poda, y por lo tanto ninguna propagación.

Tabla 4.3: Iteraciones efectuadas por 2-C3OP para el ejemplo de la Figura 4.1.

Iter	(C_i, t)	*change*	*AddQ*
1	$(C_{02}, 1)$	0	
2	$(C_{12}, 1)$	3	$(C_{02}, 1)$
3	$(C_{02}, 1)$	1	

Tabla 4.4: Iteraciones efectuadas por el procedimiento *Revise 2-C3OP* para el ejemplo de la Figura 4.1.

Iter 1. $(C_{02}, 1)$

D_0 a	D_2 b	R_{ij}	valido	$sInv$ $[b]$	$minS$	ch_i	Poda X_i
0	0	R_{02}	si	1	0		
1	0	R_{02}	no				
	1	R_{02}	si	1			
2	0	R_{02}	no				
	1	R_{02}	no				
	2	R_{02}	si	1			

$minS$	Valor b	Recibe $sInv[b]$	R_{ij}	valido	Retorna $sInv[b]$	ch_j	Poda X_j
→	0	1			0		
	1	1			0		
	2	1			0		

Iter 2. $(C_{12}, 1)$

D_1 a	D_2 b	R_{ij}	valido	$sInv$ $[b]$	$minS$	ch_i	Poda X_i
0	0	R_{12}	si		1		
		$R_{12'}$	no				
	1	R_{12}	si				
		$R_{12'}$	si	1			
1	0	R_{12}	no				
	1	R_{12}	si				
		$R_{12'}$	no				
	2	R_{12}	si				
		$R_{12'}$	si	1			
2	0	R_{12}	no				
	1	R_{12}	no				
	2	R_{12}	si				
		$R_{12'}$	no			1	2

$minS$	Valor b	Recibe $sInv[b]$	R_{ij}	hold	Retorna $sInv[b]$	ch_j	Poda X_j
	0	0			0	2	0
→	1	1			0		
	2	1			0		

Iter 3. $(C_{02}, 1)$

D_0 a	D_2 b	R_{ij}	hold	$sInv$ $[b]$	$minS$	ch_i	Poda X_i
0	1	R_{02}	no				
	2	R_{02}	no			1	0
1	1	R_{02}	si	1	1		
2	1	R_{02}	no				
	2	R_{02}	si	1			

$minS$	Valor b	Recibe $sInv[b]$	R_{ij}	hold	Retorna $sInv[b]$	ch_j	Poda X_j
→	1	1			0		
	2	1			0		

Leyenda: $sInv \equiv suppInv; minS \equiv minSupp; ch_i \equiv change_i; ch_j \equiv change_j$

4.3. 2-C3OPL

El algoritmo 2-C3OPL [16] es un algoritmo de grano grueso que alcanza la 2-consistencia en CSPs binarios y no-normalizados (ver Algoritmo 29). Este algoritmo trabaja con conjuntos de restricciones como 2-C3 [7], pero sólo requiere guardar la mitad de los conjuntos de restricciones en Q. Adicionalmente, 2-C3OPL evita chequeos de restricciones inefectivos por almacenar el último soporte encontrado como AC2001/3.1.

Así, el rendimiento de 2-C3OPL es debido a: 1) el algoritmo 2-C3OPL realiza los chequeos de restricciones bidireccionalmente; 2) almacena bidireccionalmente los

soportes para cada bloque de restricciones y 3) realiza inferencia (como AC7) para evitar chequeos de restricciones innecesarios. No obstante, la inferencia es realizada utilizando estructuras de datos (*suppInv*, *minSupp* y t) que son compartidas por todas las restricciones, y la matriz (*Last*) donde sus valores son compartidos por todas las restricciones pertenecientes al conjunto de restricciones.

Las estructuras *suppInv*, *minSupp* y t son las mismas que las utilizadas en 2-C3OP.

Algoritmo 29: Procedimiento 2-C3OPL

```
   Datos: Un CSP, P = ⟨X, D, C⟩ donde C es el conjunto de restricciones C_ij /*C_ij contiene
          únicamente restricciones directas R_ij */
   Resultado: verdadero, matrizLast y P' (el cual es 2-consistente) o  falso y P' (el cual es
              inconsistente porque se ha vaciado un dominio)
 1 principio
 2     para cada conjunto C_ij in C hacer
 3         Q ← Q ∪ {(C_ij, t) : t = 1}
 4         Last[C_ij, Xi, a] ← dummy_value; Last[C_ij, Xj, b] ← dummy_value; ∀a ∈ D_i; ∀b ∈ D_j
 5     para cada d ∈ D_max hacer
 6         suppInv[d] ← 0
 7     mientras Q ≠ ∅ hacer
 8         Seleccione y elimine (C_ij, t) de la cola Q con t = {1, 2, 3}
 9         change ← ReviseOPL((C_ij, t))
10         si change > 0 entonces
11             si change ≥ 1 ∧ change ≤ 3 entonces
12                 Q ← Q ∪ AddQ(change, (C_ij, t))
13             sino
14                 retornar falso /*dominio vacío*/
15     retornar verdadero
16 fin
```

Inicialmente, el procedimiento 2-C3OPL almacena en la cola Q los conjuntos de restricciones $(C_{ij}, t) : t = 1$, inicializa el vector *suppInv* a cero y la matriz *Last* a un valor *dummy_value*. Luego, entra en un ciclo donde se selecciona y se revisa cada bloque de restricciones almacenados en Q, hasta que no ocurran cambios (Q se ha vaciado), o hasta que se vacíe el dominio de una variable. El primer caso asegura que cada valor del dominio es 2-consistente y el segundo caso indica que el problema es inconsistente.

El procedimiento ReviseOPL (ver Algoritmo 30) utiliza tres variables internas: $change_i$, $change_j$ y *minorSupp*. Las variables $change_i$ y $change_j$ son inicializadas a cero y son usadas para recordar a que dominios se les realizó poda. Para ello, utiliza las siguientes reglas:

- Si el dominio D_i fue podado, entonces $change_i = 1$.
- Si el dominio D_j fue podado, entonces $change_j = 2$.
- Si ambos dominios fueron podados, entonces $change_i = 1$ y $change_j = 1$.

Posteriormente, son sumadas en la variable *change*, donde $change = change_i + change_j$. La variable *minorSupp* es inicializada a un valor *dummy_value*.

Durante el ciclo de pasos del 3-11, cada valor en D_i es chequeado[2]. En primer lugar, se chequea la matriz *Last*. Si el valor almacenado en $Last[C_{ij}, X_i, a]$ pertenece al dominio D_j, no se necesita hacer un chequeo de restricción porque este valor $a \in D_i$ tiene un soporte b aún valido en el dominio D_j (dicho soporte b fue encontrado en una iteración previa). Además, la variable *minSupp* es actualizada. Si el valor almacenado en la matriz $Last[C_{ij}, X_i, a]$ no pertenece al dominio D_j, entonces es buscado un soporte $b' \in D_j$. Si el valor $b' \in D_j$ soporta al valor $a \in D_i$, entonces $suppInv[b'] = a$ por la simetría de la restricción. Adicionalmente, el primer valor $b \in D_j$ (que soporta a un valor en D_i) es almacenado en la variable *minSupp* y la matriz *Last* es actualizada bidireccionalmente.

La segunda parte del Algoritmo 30 es realizada en función de los valores que posean las variables t y $change_i$.

Si $t = 2$ o $t = 3$, y $change_i = 0$, entonces C_{ij} no requiere ser chequeada debido a que la restricción, en el ciclo anterior, no ha generado ninguna poda. Sin embargo, si $t = 1$ entonces C_{ij} requiere una evaluación bidireccional completa. Si $t = 2$ o $t = 3$, y $change_i = 1$, entonces C_{ij} también requiere de una evaluación bidireccional completa. En ambos casos, el procesamiento de *suppInv* se puede hacer de tres formas diferentes: 1) los valores $b \in D_j : b < minSupp$ son podados sin realizar ningún chequeo. Además, la variable $change_j$ es actualiza a $change_j = 2$ para indicar que ocurrió un cambio en el segundo ciclo. 2) Los valores de b con $suppInv[b] > 0$ no serán chequeados debido a que ellos ya están soportados en D_i. En consecuencia, son inicializados a 0 para la posterior utilización de este vector. 3) Los valores b con $b > minSupp$ y $suppInv[b] = 0$ serán chequeados según dos casos: (i) es encontrado un soporte a y la matriz *Last* es actualizada bidireccionalmente[3] o en otro caso (ii)

[2]Si $t = 2$, entonces es usado el operador inverso.

[3]Observe que la matriz *Last* no almacena el soporte más pequeño, sino el último soporte encontrado.

serán eliminados. En este último caso, la variable es actualizada a $change_j = 2$ para indicar que han ocurrido cambios (poda de valores) en el segundo ciclo.

Algoritmo 30: Procedimiento ReviseOPL

Datos: Un *CSP* P' definido por dos variables $X = (X_i, X_j)$, dominios D_i y D_j, tupla (C_{ij}, t), matriz *Last* y el vector *suppInv*.
Resultado: dominios D_i y D_j actualizados, de tal forma que: X_i es 2-consistente en relación a X_j y D_j, y que también X_j es 2-consistente en relación a X_i o ($D_i = \emptyset$ o $D_j = \emptyset$; y, la variable entera *change*

```
1  principio
2      change_i = 0; change_j = 0
3      minSupp = dummy_value
4      para cada a ∈ D_i hacer
5          si valor almacenado en Last[C_ij, X_i, a] ∈ D_j entonces
6              Verificar y actualizar minSupp con el valor almacenado en Last[C_ij, X_i, a]
7              Siguiente valor a
8          si ∄b ∈ D_j de forma que (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ (C_ij, t) entonces
9              Eliminar a de D_i; change_i ← 1
10         sino
11             Last[C_ij, X_i, a] ← b; Last[C_ij, X_j, b] ← a; suppInv[b] ← 1
12             Verificar y actualizar minSupp con el valor b
13     si ([(t = 2 ∨ t = 3) ∧ change_i = 1] ∨ t = 1) entonces
14         para cada b ∈ D_j hacer
15             si b < minSupp entonces
16                 Eliminar b de D_j; change_j ← 2
17             sino
18                 si suppInv[b] > 0 entonces
19                     suppInv[b] ← 0
20                 sino
21                     si ∄a ∈ D_i de tal forma que (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ (C_ij, t) entonces
22                         Eliminar b de D_j; change_j ← 2
23                     sino
24                         Last[C_ij, X_j, b] ← a
25     change ← change_i + change_j
26     retornar change
27 fin
```

El procedimiento *AddQ* (ver Algoritmo 31) añadirá las tuplas que requieren ser re-evaluadas. La inclusión de tuplas dependerá de las tuplas que estén almacenadas en la cola Q y de la variable *change*. En función del valor que tenga la variable t, generada por el procedimiento *AddQ*, el procedimiento *Revise* guiará la búsqueda. Si $t = 1$, la búsqueda será bidireccional y completa (directa e inversa); si $t = 2$, la búsqueda será inversa y, se hará búsqueda directa, si y sólo si se efectúa alguna poda (en la búsqueda inversa); finalmente, si $t = 3$ la búsqueda será directa, y se hará búsqueda inversa, si y sólo si se ha efectuado alguna poda en la búsqueda

directa.

Algoritmo 31: Procedimiento AddQ

```
   Datos: Variable entera change, tupla (C_ij, t) y cola Q.
   Resultado: Cola Q actualizada
 1 principio
 2     si ((change = 1 ∧ t = 2) ∨ (change = 2 ∧ t = 1)) entonces
 3         c ← j
 4     si ((change = 2 ∧ t = 2) ∨ (change = 1 ∧ t = 1)) entonces
 5         c ← i
 6     si (change = 3) entonces
 7         c ← j y o ← j
 8     para cada C_kl con C_kl ≠ C_ij hacer
 9         si change = 3 entonces
10             Q ← Q ∪ {(C_kl, 1)|(c = l ∨ o = k) ∨ (c = k ∧ o ≠ l) ∨ (c ≠ k ∧ o = l)}
11         sino
12             si ((t = 2 ∧ c = k) ∨ (t = 1 ∧ c = l)) entonces
13                 Q ← Q ∪ {(C_kl, 3)|[(C_kl, 1) ∨ (C_kl, 3)] ∉ Q}
14             si ((t = 2 ∧ c = l) ∨ (t = 1 ∧ c = k)) entonces
15                 Q ← Q ∪ {(C_kl, 2)|[(C_kl, 1) ∨ (C_kl, 2) ∨ (C_kl, 3)] ∉ Q}
16 fin
```

4.4. 2-C4

Durante el proceso de análisis de AC4, pudo observarse que:

1. Cada vez que la fase de inicialización de AC4 evalúa cada restricción, almacena la información de los valores de las variables y de los soportes.

2. La restricción directa R_{ij} y la restricción inversa R'_{ij} comparten las mismas variables (X_i y X_j).

3. Por definición de simetría de la restricción, el soporte es bidireccional.

4. Cada vez que un valor $a \in D_i$ es eliminado, entonces la fase de propagación debe efectuarse sobre la tupla $\langle X_i, a \rangle$.

5. En problemas no-normalizados, pueden existir diferentes restricciones R_{ij} entre el mismo par de variables X_i y X_j.

Debido a 1 y 2, se detectan ineficiencias sobre AC4 ya que los valores para M, S y $Counter$ pueden ser actualizados para $\langle X_j, b \rangle$ cuando es evaluada la restricción

directa R_{ij}. En este punto, únicamente se han perdido los valores que podrían ser podados en X_j (si existieran) porque el ciclo interno es ejecutado varias veces (es decir, una vez por cada valor $a \in D_i$). Si existe un soporte, se realiza una actualización de S. Sin embargo, AC4 únicamente actualiza esta estructura para la variable X_i y es ignorado el hecho de que puede actualizar la variable X_j (item 3). Como se indica en 4, si no hay un soporte para la tupla $\langle X_i, a\rangle$ porque $total = 0$, se realiza una propagación de dicha tupla en Q. Pero la propagación sólo será efectiva si la eliminación del valor a puede afectar la consistencia de una o más variables evaluadas. Esta información está almacenada en S. Si S está vacía es debido a que el valor $a \in D_i$ eliminado no soporta ningún otro valor. No obstante, esto es ignorado en AC4 y hace que genere una propagación ineficiente. Finalmente, como el item 5 indica, restricciones diferentes R_{ij} pueden generar diferentes cantidades de soportes.

2-C4 es un algoritmo de grano fino que alcanza la 2-consistencia en CSPs binarios no-normalizados (ver Algoritmo 33). Este algoritmo trabaja con conjuntos de restricciones como 2-C3 [7] y 2-C3OP [8], con las siguientes dos variantes: sólo requiere almacenar las restricciones directas en Q (como 2-C3OP) y cada $C_{ij} \in C$ es evaluada una única vez (como AC4 evalúa cada $R_{ij} \in R$).

Por lo tanto, el aumento de rendimiento de 2-C4, en general, se debe: 1) el algoritmo 2-C4 realiza chequeos bidireccionalmente; 2) sólo se consideran soportes válidos aquellos que satisfacen a todo el conjunto de restricciones C_{ij}; 3) almacena bidireccionalmente los soportes encontrados para cada C_{ij}; y 4) realiza inferencia para evitar chequeos ineficientes. No obstante, la inferencia es realizada utilizando la estructura $suppInv$ que es compartida por todas las restricciones.

Para poder realizar una única vez el chequeo de restricciones de C e identificar los valores relevantes que requerirán ser re-evaluados, 2-C4 usa las mismas estructuras que AC4, pero lo hace sobre los conjuntos de restricciones y no sobre las restricciones del problema. También añade un vector, denominado $suppInv$, para almacenar los soportes inversos de cada valor de una variable. Así, en la revisión de cada valor $a \in D_i$ son actualizados $Counter[X_i, a, X_j]$, y posterior a ello, se pueden eliminar los valores $b \in D_j$ cuyo $suppInv[b] = 0$ (si los hubiere). De esta forma, no es requerida evaluar la restricción inversa R'_{ji}.

Las estructuras de datos, requeridas por 2-C4, son las siguientes:

- **S** es una matriz $S[X_j, b]$ que contiene la lista de pares $\langle X_i, a\rangle$ tales que $\langle X_j, b\rangle$

es su soporte. Se debe tener en cuenta que el mismo par $\langle X_i, a\rangle$ puede aparecer más de una vez en S.

- **Counters** es una matriz $Counter[X_i, a, X_j]$ que contiene el número de soportes para el valor $a \in D_i$ en la variable X_j.
- **M** es una matriz $M[X_i, a]$ que almacena el valor 1 si el valor $a \in D_i$ o almacena el valor 0 si el valor $a \notin D_i$ (indicando que la tupla $\langle X_i, a\rangle$ ha sido eliminada).
- **Q** es una cola que almacena tuplas $\langle X_i, a\rangle$ (valores eliminados) en espera de un procesamiento posterior.
- **suppInv** es un vector de enteros cuyo tamaño es el valor de la talla máxima del dominio de las variable($maxD$). Almacena un valor mayor o igual a uno 1 cuando el valor X_j está soportado.

El cuerpo principal del algoritmo 2-C4 tiene dos fases: i) Inicialización de las estructuras de datos y ii) Propagación. Las inicializaciones son utilizadas para recordar pares de variable-valor consistentes (matriz S), contar la cantidad de soportes (matriz $Counter$), eliminar los valores no soportados y recordar dichas eliminaciones (matriz M y cola Q, respectivamente).

El procedimiento *Initialize2C4* inicializa la cola Q a cero, la matriz S a nulo, la matriz M a uno, la matriz $Counter$ a cero y el vector $suppInv$ a cero. Luego, son efectuados cuatro ciclos para seleccionar y revisar el bloque de restricciones $R_{ij} \in C_{ij}$ para cada valor $a \in D_i$ y para cada valor $b \in D_j$.

La poda se realiza de acuerdo a los contadores en cada conjunto de restricciones. El contador de soportes de la variable X_i ($total$) es inicializado a 0 antes de evaluar cada valor de $a \in D_i$. No obstante, el contador de soportes de la variable X_j (soportes inversos) debe ser dividido en dos contadores diferentes: $Counter[X_j, b, X_i]$ y $suppInv[b]$. El vector $suppInv$ almacena el número de soportes de cada valor de X_j. Este vector es inicialicializado a cero (ver Algoritmo 32, paso 6). Cuando el valor $b \in D_j$ soporta el valor $a \in D_i$, $suppInv[b]$ será incrementado en 1 (ver Algoritmo 32, paso 22). Durante el ciclo de pasos 8-30, este vector es actualizado para ser analizado en el paso 34. Posteriormente al procesamiento de todos los valores de D_i, si un valor b de D_j no tiene soportes ($suppInv[b] = 0$), entonces este valor b es eliminado de D_j. Si $suppInv[b] > 0$, entonces b está soportado y ese elemento del

Algoritmo 32: Procedimiento Initialize2C4

Datos: Un CSP, $P = \langle X, D, C\rangle$ donde C es el conjunto de restricciones C_{ij}
Resultado: initial=**verdadero** y P', Q, S, M, $Counter$ o initial=**falso** y P' (el cual es inconsistente porque se ha vaciado un dominio).

```
1  principio
2      Q ← {}
3      S[X_j, b] ← {}   /* ∀X_j ∈ X ∧ ∀b ∈ D_j */
4      M[X_i, a] ← 1   /* ∀X_i ∈ X ∧ ∀a ∈ D_i */
5      Counter[X_i, a, X_j] ← 0   /* ∀X_i, X_j ∈ X ∧ ∀a ∈ D_i */
6      suppInv[b] ← 0   /* ∀b ∈ [1, maxD] */
7      para cada conjunto C_ij ∈ C hacer
8          para cada a ∈ D_i hacer
9              total ← 0
10             para cada b ∈ D_j hacer
11                 R_ij ← primera R_ij ∈ C_ij
12                 supported ← 1
13                 mientras supported = 1 ∧ R_ij ≠ NULL hacer
14                     si (⟨X_i, a⟩, ⟨X_j, b⟩) ∈ R_ij entonces
15                         R_ij ← siguiente R_ij ∈ C_ij
16                     sino
17                         supported ← 0
18                 si supported = 1 entonces
19                     total ← total + 1
20                     Añadir (S[X_j, b], ⟨X_i, a⟩)
21                     Counter[X_j, b, X_i] ← Counter[X_j, b, X_i] + 1
22                     suppInv[b] ← suppInv[b] + 1
23                     Añadir(S[X_i, a], ⟨X_j, b⟩)
24             si total = 0 entonces
25                 Eliminar a de D_i
26                 M[X_i, a] ← 0
27                 si S[X_i, a] ≠ {} entonces
28                     Q ← Q ∪ ⟨X_i, a⟩
29             sino
30                 Counter[X_i, a, X_j] ← total
31         si D_i = φ entonces
32             retornar initial ← falso
33         para cada b ∈ D_j hacer
34             si suppInv[b] = 0 entonces
35                 Eliminar b de D_j
36                 M[X_j, b] ← 0
37                 si S[X_j, b] ≠ {} entonces
38                     Q ← Q ∪ ⟨X_j, b⟩
39             sino
40                 suppInv[b] ← 0
41         si D_j = φ entonces
42             retornar initial ← falso
43     retornar initial ← verdadero y Q, M, S, Counter
44 fin
```

Algoritmo 33: Procedimiento 2-C4

```
Datos: Un CSP, P = ⟨X, D, C⟩ donde C es el conjunto de restricciones no-normalizadas C_ij /*C_ij
       contiene únicamente restricciones directas R_ij */
Resultado: verdadero y P' (el cual es 2-consistente) o  falso y P' (el cual es inconsistente)
 1  principio
 2      Initialize2C4(P)
 3      si initial = verdadero entonces
 4          mientras Q ≠ φ hacer
 5              Seleccione y elimine ⟨X_j, b⟩ de la cola Q
 6              para cada tupla ⟨X_i, a⟩ ∈ S[X_j, b] hacer
 7                  Counter[X_i, a, X_j] ← Counter[X_i, a, X_j] − 1
 8                  si Counter[X_i, a, X_j] = 0 ∧ M[X_i, a] = 1 entonces
 9                      Eliminar a de D_i
10                      si D_i = φ entonces
11                          retornar falso
12                      sino
13                          si S[X_i, a] ≠ {} entonces
14                              Q ← Q ∪ ⟨X_i, a⟩
15                      M[X_i, a] ← 0
16          retornar verdadero
17      sino
18          retornar falso
19  fin
```

vector es inicializado a 0 para utilizarlo en el siguiente conjunto de restricciones (ver Algoritmo 32, pasos 33 al 40).

Para alcanzar la 2-consistencia, 2-C4 únicamente computa un soporte si la instanciación $\langle X_i, a\rangle$ y $\langle X_j, b\rangle$ satisfacen todas las restricciones $R_{ij} \in C_{ij}$. Esto es logrado inicializando la variable *supported* a 1 que actuará como una bandera y 2-C4 detendrá la revisión de C_{ij} cuando las tuplas $\langle X_i, a\rangle$ y $\langle X_j, b\rangle$ no satisfagan a la restricción R_{ij}, cambiando la variable *supported* a 0 (ver Algoritmo 32, pasos 12 a 17).

Además, 2-C4 únicamente propaga aquellas tuplas que sirven de soporte a otras tuplas. (Ver pasos 27-28 y 37-38 del Algoritmo 32 y pasos 13-14 del Algoritmo 33). Así, 2-C4 evita propagaciones ineficientes de tuplas en Q y su posterior chequeo ineficiente. El proceso es detenido cuando un dominio queda vacío (Algoritmo 32, pasos 31 y 41, y Algoritmo 33, paso 10).

Correctitud y complejidad de 2-C4

1. El algoritmo 2-C4 es correcto.

   Prueba: Por contradicción, supongamos que un valor $c \in D_i$ es eliminado

para X_i pero este tiene un soporte con todos los valores de las variables con las que está restringido X_i. El valor $c \in D_i$ podría haber sido eliminado en la fase de inicialización (Inizialize2C4) o en el paso 9 de 2-C4. Estudiemos ambos casos:

- Si el valor $c \in D_i$ es eliminado en la fase Inizialize2C4, entonces es eliminado en el paso 25 o en el 35.

  Si c es eliminado en el paso 25, entonces una restricción directa se está analizando y $total = 0$ por lo ningún valor en X_j es soporte de $c \in D_i$. $\#Contradicción$

  Si c es eliminado en el paso 35, entonces una restricción inversa se está analizando y $suppInv[c] = 0$ de forma tal que b no es un soporte de ningún valor. $\#Contradicción$

- Si el valor $c \in D_i$ es eliminado en el paso 9 de 2-C4, entonces es debido a que $Counter[X_i, c, X_j] = 0$, de manera que $X_i = c$ no es soporte de la variable X_j. $\#Contradicción$

De esta forma, cada valor $c \in D_i$ eliminado de X_i por 2-C4 no está soportado por ninguno de los valores de las variables con las que X_i está restringido, así este valor c no formará parte de ninguna solución.

2. La complejidad de 2-C4 es $O(fd^2)$, donde f es el número de conjuntos de restricciones binarias y d la talla del dominio del problema.

   Prueba: La fase la fase de inicialización de 2C4 tiene un coste temporal de $O(fd^2)$. El paso 7 analiza cada conjunto en $(O(f))$ y, para cada set es analizado cada valor $a \in D_i$, (paso 8), y para cada valor $a \in D_i$ cada valor $b \in D_j$ es nuevamente analizado (paso 10), por lo que el coste actual es $O(fdd) = O(fd^2)$. A continuación 2-C4 hace un ciclo para la cola Q que almacena las tuplas que están esperando su re-procesamiento. La cardinalidad de Q es $|Q| = nd$ donde n es el número de variables. Si asumimos que $n < fd$ entonces la complejidad de 2-C4 es $O(fd^2 + nd) < O(fd^2 + fd^2) \equiv O(fd^2)$.

4.5. 2-C6

A continuación, presentamos el algoritmo 2-C6, el cual alcanza la 2-consistencia en CSP binarios no-normalizados (ver Algoritmo 34). Este algoritmo es una reformulación del algoritmo AC6 [29]. 2-C6 trabaja con conjuntos de restricciones, como 2-C3 [7], pero en vez de propagar conjuntos de restricciones, propaga tuplas $\langle variable, valor \rangle$. Esto lo convierte en un algoritmo de grano-fino. 2-C6 mantiene estructuras de datos más ligeras que 2-C4 y sólo almacena un único soporte. El algoritmo 2-C6 requiere estructuras similares a las que utiliza AC6, excepto en la realización del chequeo de restricciones de C y en identificar los valores relevantes que requerirán ser re-evaluados. En estos casos, 2-C6 lo hace sobre los conjuntos de restricciones y no sobre las restricciones del problema, con lo cual logra alcanzar la 2-consistencia.

Algoritmo 34: Procedimiento 2-C6

```
   Datos: Un CSP P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es 2-consistente) o  falso y P' (el cual es inconsistente)
 1 principio
 2     Initialize2C6(P)
 3     si initial = verdadero entonces
 4         mientras Q ≠ φ hacer
 5             Seleccione y elimine ⟨X_j, b⟩ de la cola Q
 6             para cada ⟨X_i, a⟩ ∈ S[X_j, b] hacer
 7                 Elimine ⟨X_i, a⟩ de S[X_j, b]
 8                 si M[X_i, a] = 1 entonces
 9                     c ← b
10                     C_ij ← findSetC(X_i, X_j)
11                     emptysupport ←nextSupport2C6(C_ij, a, c)
12                     si emptysupport =verdadero entonces
13                         Eliminar a de D_i
14                         M[X_i, a] ← 0
15                         Q ← Q ∪ ⟨X_i, a⟩
16                         si D_i = φ entonces
17                             retornar falso
18                     sino
19                         Añadir (S[X_j, c], ⟨X_i, a⟩)
20         retornar verdadero
21     sino
22         retornar falso
23 fin
```

En cuanto a su funcionamiento, 2-C6 (ver Algoritmo 34) consta de dos fases: inicialización y propagación. En la fase de inicialización (ver Algoritmo 35) son

almacenadas las restricciones no-normalizadas R_{ij} en conjuntos C_{ij} (pasos del 5 al 8); seguidamente, los C_{ij} y C'_{ji} son almacenados en C (pasos 9 y 10). Luego, en un bucle, para cada conjunto C_{ij} es buscado un soporte b para cada valor a de cada variable X_i sobre el conjunto de restricciones C_{ij} en las que está vinculada (pasos 11 al 22). El soporte buscado siempre es el valor más pequeño, para lo cual InitializeAC6 llama al Algoritmo 36. Si dicho soporte existe, este es almacenado en la estructura S (ver Algoritmo 35, paso 20). Por el contrario, si no es posible encontrar dicho soporte b, el valor $a \in D_i$ es eliminado y la tupla $\langle X_i, a\rangle$ es almacenada en la lista Q para su posterior procesamiento (ver Algoritmo 35, pasos 15 al 18). El proceso se repite hasta que: a) se hayan revisado todos los conjuntos de restricciones continuando con la fase de propagación o b) se haya vaciado el dominio de una variable, indicando de esta forma que el problema es inconsistente.

El procedimiento *nextSupport2C6* de 2-C6 (ver Algoritmo 36) es el encargado de buscar un soporte en D_j a cada valor de la variable X_i. Es muy similar al procedimiento *nextSupport* de AC6, la diferencia se centra en que la instanciación $\langle X_i = a, X_j = b\rangle$ debe ser verificada con el bloque de restricciones C_{ij} en vez de hacerlo con una única restricción R_{ij}. El conjunto de restricciones C_{ij} puede también ser ordenado evitando así chequeos innecesarios. Si ordenamos este conjunto desde las restricciones más restrictivas a las menos restrictivas, el chequeo de las restricciones podrá encontrar antes las inconsistencias y evitar así que se lleven a cabo chequeos innecesarios.

En la fase de propagación (ver Algoritmo 34, pasos 4 al 20), se ejecuta un bucle donde es extraída una tupla $\langle X_j, b\rangle$ de la cola Q para luego procesar la consecuencia de su eliminación. Así, para cada una de las tuplas $\langle X_i, a\rangle$ contenidas en $S[X_j, b]$, si el valor a está aún presente en el dominio de X_i (ver Algoritmo 34, paso 8), es buscado un nuevo soporte $c \in D_j$ que satisfaga el conjunto de restricciones C_{ij}. De no conseguirse, el valor a es eliminado de D_i y una tupla $\langle X_i, a\rangle$ es añadida a Q (ver Algoritmo 34, pasos 12 al 17). Por el contrario, si es conseguido el nuevo valor de $c \in D_j$ pasará a formar soporte de a y será recordado en S (ver Algoritmo 34, paso 19). Este proceso continuará hasta que todas las tuplas almacenadas en Q sean procesadas o cuando se torne vacío el dominio de una variable. En el primer caso, el algoritmo indica que el problema es 2-consistente y en el segundo indica que es inconsistente.

Algoritmo 35: Procedimiento Initialize2C6

Datos: Un CSP $P = \langle X, D, R\rangle$ /*R contiene las restricciones directas e inversas*/
Resultado: Variable Booleana *initial* (donde *initial* =**verdadero** indica que ningún dominio se ha vaciado e *initial* =**falso** indica que P' es inconsistente); cola Q; matrices S y M; y CSP P' actualizados.

```
1  principio
2      Q ← {}
3      S[X_j, b] ← {}  /* ∀X_j ∈ X ∧ ∀b ∈ D_j */
4      M[X_i, a] ← 1  /* ∀X_i ∈ X ∧ ∀a ∈ D_i */
5      para cada i, j hacer
6          C_ij ← ∅
7      para cada arco R_ij ∈ R hacer
8          C_ij ← C_ij ∪ R_ij
9      para cada conjunto C_ij hacer
10         C ← C ∪ {C_ij, C'_ji}
11     para cada conjunto C_ij ∈ C hacer
12         para cada valor a ∈ D_i hacer
13             b ← dummy_value
14             emptysupport ←nextSupport2C6(C_ij, a, b)
15             si (emptysupport =verdadero) entonces
16                 Eliminar a de D_i
17                 M[X_i, a] ← 0
18                 Q ← Q ∪ ⟨X_i, a⟩
19             sino
20                 Añadir (S[X_j, b], ⟨X_i, a⟩)
21             si D_i = φ entonces
22                 retornar initial ← falso
23     retornar initial ← verdadero
24 fin
```

Algoritmo 36: Procedimiento nextSupport2C6

Datos: Conjunto de restricciones C_{ij}, valor $a \in D_i$ y valor $b \in D_j$
Resultado: Variable Booleana *emptySupport* (donde *emptySupport* =**verdadero** indica que no ha encontrado un valor de soporte b en el dominio D_j para el valor a y *emptySupport* =**falso** indica que ha encontrado un valor de soporte); valor b actualizado.

```
1  principio
2      si (b ≤ que valor más grande de D_j) entonces
3          emptySupport ← falso
4          b ← siguiente valor de D_j
5          mientras ⟨a, b⟩ ∉ C_ij ∧ emptySupport =falso hacer
6              si (b < que valor más grande de D_j) entonces
7                  b ← siguiente valor de D_j
8              sino
9                  emptySupport ← verdadero
10     sino
11         emptySupport ← falso
12     retornar emptySupport y b
13 fin
```

4.6. AC3NH

Para poder aplicar a problemas binarios no-normalizados los algoritmos de consistencia existentes en la literatura, es necesario normalizar las restricciones no-normalizadas. En el Capítulo 2, presentamos teóricamente cómo transformar un problema no-normalizado en uno normalizado, utilizando la representación extensional de las restricciones y la intersección de tuplas. Con el algoritmo AC3NH desarrollamos un esquema de normalización híbrido, que permite medir el coste de la normalización en tiempo y número de chequeos de restricciones. AC3NH es capaz de detectar cuáles son las restricciones no-normalizadas, las normaliza (las transforma en tuplas permitidas) y deja igual el resto de restricciones normalizadas (representación intensional), para posteriormente aplicar AC3 y alcanzar así la arco-consistencia.

El algoritmo AC3NH [14] trabaja de forma similar al algoritmo AC3, ya que después de invocar al proceso que almacena las restricciones (procedimiento *KeepConstraints*, ver Algoritmo 38) y generar los nodos en Q, en un ciclo extrae los nodos almacenados en Q, revisa que existan los soportes de las variables involucradas, elimina los valores no soportados y propaga en Q las consecuencias de la eliminación de valores en los dominios. Esto ocurre hasta que no existan más cambios ($Q = \emptyset$) o hasta que el dominio de una variable se torne vacío.

Los procedimientos *KeepConstraints* y *Normalize* (ver Algoritmos 38 y 39, respectivamente), generan las estructuras que permiten detectar si el problema es no-normalizado antes de generar los nodos en Q.

Los nodos almacenados en Q son tuplas de la forma (C, s), donde s es un número entero que puede tomar los valores $1, 2$ ó 3 determinando como se procesará la restricción almacenada. C es una estructura que contiene los siguientes campos:

- C_{id} un identificador del nodo;
- C_i almacena la variable X_i de la restricción R_{ij} y permite acceder a su dominio D_i;
- C_j almacena la variable X_j de la restricción R_{ij} y permite acceder a su dominio D_j;
- C_R es una lista que almacena una o más restricciones $R_{ij} \in R$. De ser más

Algoritmo 37: Procedimiento AC3NH

```
   Datos: un CSP, P = ⟨X, D, R⟩
   Resultado: verdadero y P' (el cual es 2-consistente) o falso y P' (el cual es inconsistente
              porque se ha vaciado un dominio)
 1 principio
 2     Q ← ∅
 3     sListaR ← ∅
 4     seguir ←KeepConstraints(P, sListaR, Q)
 5     si seguir = verdadero entonces
 6         mientras Q ≠ ∅ hacer
 7             Seleccione y elimine el primer nodo (C, s) de la cola Q con s ∈ {1, 2, 3}
 8             si s = 1 ∨ s = 2 entonces
 9                 agregar ←Revise(C.R_ij, s)
10             sino
11                 agregar ←ReviseExtensional(C.C_tuplas, D_i, D_j, s)
12             si agregar = 5 entonces
13                 retornar falso /*dominio vacío */
14             sino
15                 si agregar > 1 entonces
16                     AddPropagations(Q, (C, s), sListaR, agregar)
17     retornar seguir
18 fin
```

de una restricción, las restricciones almacenadas en C_R serán utilizadas en el proceso de Normalización;

- C_{numRes} indica cuantas restricciones están almacenadas en C_R. Si $C_{numRes} > 1$ se activará el proceso de Normalización;
- C_{tuplas} contendrá las tuplas obtenidas del proceso de Normalización.

Si $s = 1$, indica que la restricción almacenada en C_R es intensional y se evaluará de forma directa, es decir, se evaluará un arco en sentido $i \rightarrow j$ y además indica que su propagación (en caso de efectuar podas en D_i) se hará igual que en AC3. Si $s = 2$, indica que la restricción almacenada en C_R es intensional y se evaluará de forma inversa, es decir, se evaluará un arco en sentido $j \rightarrow i$; y que su propagación (en caso de efectuar podas en D_j) se hará igual que en AC3. Por último, en el caso de que $s = 3$, indica que la restricción es extensional, por lo que hay que verificar que los dominios D_i y D_j y las tuplas almacenadas en C_{tuplas} se satisfagan.

El procedimiento *KeepConstraints* (ver Algoritmo 38) realiza tres procesos:

1. Leer y clasificar las restricciones del problema (ver Algoritmo 38, pasos 2 al

Algoritmo 38: Procedimiento KeepConstraints

Datos: Un CSP , $P = \langle X, D, R \rangle$, un vector $sListaR$ y una cola Q
Resultado: **verdadero**, $SListaR$ normalizada y cola Q actualizada o **falso** (se ha detectado inconsistencia porque se ha vaciado un dominio)

```
1  principio
2      esNormalizado ← verdadero
3      para cada arco R_ij ∈ R hacer
4          si sListaR = ∅ entonces
5              C = newNodo(C_id = id_R, C_i = X_i, C_j = X_j, C_R = R_ij, C_numRes = 1, C_tuplas = ∅)
6              Añadir (C, sListaR)
7          sino
8              si (∃ C ∈ sListaR |C_f = X_i ∧ C_k = X_j donde X_i, X_j ∈ R_ij) entonces
9                  Añadir (R_ij, C_R) ∧ C_numRes ← C_numRes + 1
10                 esNormalizado ← falso
11             sino
12                 C = newNodo(C_id = id_R, C_i = X_i, C_j = X_j, C_R = R_ij, C_numRes = 1, C_tuplas =
                   ∅)
13                 Añadir (C, sListaR)
14     si esNormalizado = falso entonces
15         continuar ←Normalize(sListaR)
16         si continuar = falso entonces
17             retornar falso
18     para cada nodo C ∈ sListaR hacer
19         si C_numRes = 1 entonces
20             Añadir ((C, 1), Q) y Añadir ((C, 2), Q)
21         sino
22             Añadir ((C, 3), Q)
23     retornar verdadero
24 fin
```

13);

2. Normalizar las restricciones no-normalizadas (ver Algoritmo 38, pasos 14 al 17), que se realizará sólo si la variable $esNormalizado = falso$, ya que se ha detectado que el problema es no-normalizado (ver Algoritmo 38, paso 10) y,

3. Almacenar las restricciones a evaluar en Q (pasos 18 al 22).

Así, en el proceso de leer y clasificar las restricciones, se efectúa un ciclo donde para cada una de las restricciones $R_{ij} \in R$, se verifica la presencia de ambas variables X_i y X_j, en uno de los nodos C almacenados en el vector $sListaR$. Si no están presentes ambas variables en un nodo C, se crea y añade un nuevo nodo C a $sListaR$. Por el contrario, si se ha encontrado un nodo C que contenga ambas variables, ello significa que se han detectado que el problema es no-normalizado, con

lo cual, la restricción R_{ij} es almacenada en la estructura C_R (para su proceso posterior), se incrementa el valor de C_{numRes} y se cambia a falso la variable Booleana $esNormalizado$.

Si la variable $esNormalizado$ tiene el valor de falso, se ejecuta el proceso de Normalización (ver Algoritmo 39). Este proceso utiliza tres conjuntos: $tuplas$, para almacenar las tuplas $\langle a, b\rangle$ que satisfacen la restricción R_{ij} (tuplas válidas), $tuplasUnion$ para almacenar todas las tuplas válidas generadas para todas las $R_{ij} \in C_R$ y $tuplasInter$ para almacenar aquellas tuplas de $tuplasUnion$ presentes en todos los conjuntos $tuplas$ de cada $R_{ij} \in C_R$. La Normalización consta de un ciclo principal, donde se procesan los nodos C almacenados en $sListaR$, que tengan más de una restricción ($C_{numRes} > 1$) y de dos ciclos internos: a) pasos 4 al 19, donde se generan las tuplas válidas para la restricción ($tuplas$) y se acumulan estas tuplas en $tuplasUnion$. b) pasos 20 al 33, donde se efectúa la intersección de las tuplas presentes en todas las restricciones y se almacena en $tuplasInter$. Finalmente si el conjunto $tuplasInter$ es distinto a vacío, almacena en C_{tuplas} el conjunto $tuplasInter$.

Pese a que es un proceso de Normalización, y no poda los valores de los dominios de las variables (ello lo hace la arco-consistencia), el Algoritmo 39 tiene dos estrategias para detectar la inconsistencia: la primera, en los pasos 12 y 13: si no se generan tuplas válidas para la restricción ($tuplas = \emptyset$); y la segunda, pasos 26 y 27: si al finalizar el proceso de intersección el conjunto de $tuplasInter$ es vacío. En cualquiera de los dos casos el procedimiento *Normalize* retornará el valor de falso, y no habrá necesidad de realizar la arco-consistencia, ya que en este punto se conoce que el problema no tiene solución.

Las *propagaciones* son realizadas con el Algoritmo 40, en función del tipo de restricción (intensional o extensional) que haya realizado la poda determina las restricciones que tienen que propagarse. Así, en restricciones intensionales la propagación se hará de la misma forma que en AC3; pero en restricciones extensionales el valor de la variable *agregar* indicará si ha cambiado X_i, X_j o ambas, tomando los valores de 1, 2 o 3, respectivamente. Es por ello que al principio del Algoritmo 40, en los pasos 2-8, se determina cuál es la variable que cambia y en los pasos 9-20, se determina que restricciones serán incluidas en la propagación, para ser nuevamente re-evaluadas.

Algoritmo 39: Procedimiento Normalize

```
   Datos: sListaR
   Resultado: verdadero y sListaR normalizada o  falso
 1 principio
 2     para cada nodo C ∈ sListaR hacer
 3         si C_numRes > 1 entonces
 4             tuplasUnion ← ∅
 5             para cada R_ij almacenada en C_R hacer
 6                 tuplas ← ∅
 7                 para cada valor a ∈ D_i hacer
 8                     para cada valor b ∈ D_j hacer
 9                         si ⟨a, b⟩ ∈ R_ij entonces
10                             Añadir (⟨a, b⟩, tuplas)
11                             Añadir (⟨a, b⟩, tuplasUnion)
12                 si tuplas = ∅ entonces
13                     retornar falso /* problema inconsistente en R_ij*/
14                 sino
15                     Añadir (tuplas, R_ij.t)
16             tuplasInter ← ∅
17             para  cada t ∈ tuplasUnion hacer
18                 continue ← verdadero
19                 mientras ∃R_ij ∈ C_ij ∧ continue = verdadero hacer
20                     si t esta en R_ij.t entonces
21                         Siguiente R_ij
22                     sino
23                         continue = falso
24                 si continue = verdadero entonces
25                     Añadir (t, tuplasInter)
26             si tuplasInter = ∅ entonces
27                 retornar falso /* problema inconsistente en C_ij*/
28             sino
29                 C_tuplas ← tuplasInter
30     retornar verdadero
31 fin
```

Algoritmo 40: Procedimiento AddPropagaciones

Datos: Un CSP, $P = \langle X, D, R \rangle$
Resultado: **verdadero** y P' (que es arco-consistente) o **falso** y P' (el cual es inconsistente porque se ha vaciado un dominio)

principio
  $cJ \leftarrow \emptyset$
  si $(s = 1 \vee (s = 3 \wedge agregar = 1))$ **entonces**
    $cambia \leftarrow C_i \wedge m \leftarrow C_j$
  si $(s = 2 \vee (s = 3 \wedge agregar = 2))$ **entonces**
    $cambia \leftarrow C_j \wedge m \leftarrow C_i$
  si $(s = 3 \wedge agregar = 3)$ **entonces**
    $cambia \leftarrow C_i \wedge m \leftarrow C_j \wedge cJ \leftarrow C_j$
  para *cada* $nodo \in sListaR$ **hacer**
    si $nodo_{id} \neq C_{id}$ **entonces**
      $i \leftarrow nodo_i$
      $j \leftarrow nodo_j$
      si $nodo_{numRes} = 1$ **entonces**
        si $(((s = 1 \vee (s = 3 \wedge agregar = 1)) \wedge cambia = i) \vee (s = 3 \wedge agregar = 3 \wedge (cambia = i \vee cambia = j \vee cJ = i \vee cJ = j)))$ **entonces**
          Añadir $(Q, (C, 1))$ *sí* $(C, 1) \notin Q$
        si $(((s = 2 \vee (s = 4 \wedge agregar = 2)) \wedge cambia = j) \vee (s = 3 \wedge agregar = 3 \wedge (cambia = i \vee cambia = j \vee cJ = i \vee cJ = j)))$ **entonces**
          Añadir $(Q, (C, 2))$ *sí* $(C, 2) \notin Q$
        sino
          si $(((s = 1 \vee s = 2) \wedge (cambia = i \vee cambia = j)) \vee (s = 3 \wedge (cambia = i \vee cambia = j \vee cJ = i \vee cJ = j)))$ **entonces**
            Añadir $(Q, (C, 4))$ *sí* $(C, 4) \notin Q$
fin

4.7. Conclusiones

Analizando los algoritmos de arco-consistencia encontrados en la literatura, hemos presentado en este capítulo los seis algoritmos desarrollados que alcanzan la 2-consistencia para problemas binarios normalizados y no-normalizados: 2-C3, 2-C3OP, 2-C3OPL, 2-C4, 2-C6 y AC3NH. Los algoritmos 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6 logran mejorar en eficiencia en cuanto al número de podas, chequeos de restricciones, número de propagaciones y tiempo de cómputo tanto a los algoritmos de arco-consistencia como entre ellos mismos en uno o más parámetros, cuando se trata de problemas no-normalizados. El algoritmo AC3NH logra alcanzar la 2-consistencia, gracias al proceso de normalización que realiza. Pero, como se señaló en el Capítulo 2, el coste del proceso de normalización no es trivial, y ello es corroborado en la evaluación (Capítulos 6 y 7), donde las bondades de los algoritmos de 2-consistencia desarrollados se evidencian más aún.

Capítulo 5

Algoritmos de Búsqueda

5.1. Introducción

En los capítulos anteriores hemos presentado diversas técnicas de consistencia (encontradas en la literatura y propuestas), pero éstas están diseñadas para reducir el árbol de búsqueda, al eliminar valores que no pueden formar parte de la solución, por lo que en pocas ocasiones son capaces de encontrar una solución al problema. Debido a ello, los algoritmos de consistencia son combinados con las técnicas de búsqueda, creando así las técnicas de búsqueda híbridas (ver Figura 2.12). Como ya indicamos en el Capítulo 2, los algoritmos de búsqueda recorren el *árbol de búsqueda*, asignando valores a las variables del problema con la finalidad de encontrar una solución (o varias soluciones) al CSP o determinar que el problema no tiene solución.

Los algoritmos de búsqueda en función a su aplicabilidad, pueden clasificarse como independientes del dominio o dependientes del dominio, ya que dichos algoritmos, pueden sacrificar la generalidad de procesar cualquier dominio y explotar las características propias de un dominio en particular, que facilite el proceso de búsqueda y haga más eficiente la resolución de los problemas. Adicionalmente, en función de garantizar o no el encontrar una solución al problema, los algoritmos de búsqueda se pueden clasificar en completos (garantizan encontrar solución al problema -si tal solución existe-) o heurísticos (por sí solos no garantizan encontrar solución al problema -si tal solución existe-, pero cuando lo hacen, son muy eficientes).

Puede parecer extraño aplicar un algoritmo de búsqueda que por sí solo pueda no encontrar una solución al problema, pero en problemas reales, donde los algoritmos

de búsqueda genéricos son muy costosos en el proceso de determinar si el problema tiene o no solución, se utilizan los algoritmos heurísticos que son más eficientes.

En general, las técnicas de consistencia podan el espacio de búsqueda, pero no ofrecen una solución para el problema. De esta forma, después de aplicar una técnica de filtrado, utilizamos tres técnicas de búsqueda explicadas en el Capítulo 2: Backtracking (BT), Forward checking (FC) y Real-Full-Look-Ahead (RFLA) para el problema de asignación de horarios ferroviarios, con los siguientes resultados:

- BT es un algoritmo muy sencillo pero ineficiente debido a que: i) BT verifica la instanciación de la variable actual teniendo en cuenta sólo las asignaciones previamente realizadas. Por lo que no considera la relación de la asignación actual con las futuras asignaciones. De esta forma, la variable actual puede ser instanciada con un valor que es inconsistente con las futuras variables. El *dead-end* ocurre cuando la variable futura es instanciada. Esto podría ser evitado, si el chequeo es realizado antes, es decir, cuando la variable actual es instanciada, y de esta forma el *dead-end* podría aparecer en un estado temprano del proceso. ii) BT no recuerda acciones pasadas, lo cual ocasiona que una misma acción sea repetida innecesariamente. A medida que la complejidad del problema es mayor, la deficiencia de este algoritmo se torna más evidente.

- FC puede ser visto como la aplicación de un simple paso de arco-consistencia sobre cada restricción, que implica a la variable actual con una variable próxima, después de cada asignación de variables. Sin embargo, en dominios grandes, como lo tienen los problemas de la asignación de horarios ferroviarios, la búsqueda con FC es muy costosa en tiempo y en memoria, ya que FC debe guardar cada dominio con el fin de reutilizarlo cuando hace retrocesos.

- RFLA es más eficiente que el FC, pero como RFLA es un algoritmo *look-ahead* como FC, también su proceso de búsqueda es bastante costoso en tiempo y memoria.

En base a lo anterior, en este capítulo presentamos dos algoritmos de búsqueda desarrollados para CSPs binarios no-normalizados: el algoritmo look-**B**ack-**L**ast **S**earch (BLS) y el algoritmo SchTrains. A continuación se dará una breve explicación de ambos algoritmos:

- El algoritmo look-**Back-Last Search** (BLS) es una nueva técnica de búsqueda heurística independiente del dominio, que combina las estructuras del almacenamiento de soportes (matriz *Last*) del algoritmo de consistencia 2-C3OPL, con la técnica de búsqueda con enfoque *look-back*. BLS utiliza las estructuras generadas por 2-C3OPL para guiar el proceso de búsqueda, ahorrando tiempo de cómputo y retrocesos (*backtracks*).
- **SchTrains** es una nueva técnica de búsqueda dependiente del dominio que aprovecha los beneficios que proporcionan las técnicas de consistencia que almacenan los soportes encontrados y utiliza el conocimiento del dominio de planificación ferroviaria para realizar la búsqueda de manera eficiente.

Ya que ambos algoritmos BLS y SchTrains fueron desarrollados para resolver el problema de asignación de horarios ferroviarios, la explicación de los mismos se dará en función a este tipo de problemas[1].

5.2. Algoritmo BLS: Loop-Back Last Search

En la resolución de problemas genéricos y del problema de asignación de horarios ferroviarios, observamos lo siguiente:

- Hay algunos algoritmos de consistencia, por ejemplo AC2001/3.1 y 2-C3OPL, que almacenan los soportes encontrados en una matriz *Last*, pero esto no es explotado por los algoritmos de búsqueda publicados en la literatura.
- Las restricciones se pueden utilizar para decidir el orden de instanciación de las variables y para crear dependencias entre ellas.
- La elección de los valores de una variable dada puede basarse en ciertos criterios, en lugar de ser completamente a ciegas.
- Si un algoritmo puede reducir el número de retrocesos (backtracking) en dominios grandes y no requiere almacenar estos dominios si ocurre un retroceso, entonces este algoritmo empleará menos tiempo que otro que almacene dominios, elija cada valor a ciegas, y realize consistencia en cada paso.

[1] Los detalles y la formulación del problema de asignación de horarios ferroviarios son presentados en el Capítulo 7.

Debido a estas razones, proponemos combinar la simplicidad de los algoritmos *loop-back* con nuestro algoritmo de 2-consistencia 2-C3OPL[2], en el desarrollo del algoritmo: loop-**B**ack **L**ast **S**earch (BLS).

El algoritmo BLS, requiere de las siguientes estructuras:

- **C**: es el conjunto de bloques de restricciones no-normalizadas obtenidas del problema P, y contiene $C_{ij} \in C$, $\forall i, j$.
- **Last**: es una matriz que almacena un soporte $b \in D_j$, para cada valor $a \in D_i$ por cada conjunto de bloque de restricciones C_{ij} , $\forall i, j$.
- **listVar**: es una lista enlazada que almacena nodos $nodeVar$. La estructura de un nodo $nodeVar$ contiene los siguientes campos:
  - **var**: es un puntero que apunta a los datos relacionados con la variable $X_i \in X$. Para simplificar la codificación utilizaremos X_i en vez de var;
  - **father**: apunta a los datos relacionados con la variable X_j. Teniendo en cuenta que $X_j \in X$; $X_j \neq X_i$ y X_j puede instanciar a X_i;
  - **next**: apunta al siguiente nodo en $listVar$; y
  - **previous**: apunta al nodo previo en $listVar$.

El procedimiento BLS (ver Algoritmo 41) funciona de la siguiente forma: comienza inicializando las estructuras C y $Last$ (pasos 2 y 3 del Algoritmo 41), que comparte con el algoritmo de 2-consistencia 2-C3OPL. A continuación, se ejecuta 2-C3OPL para alcanzar la 2-consistencia, y con lo cual la matriz $Last$ queda actualizada con los soportes encontrados por el proceso de consistencia ejecutado (paso 4). Si el problema resultante P' es consistente ($continue = verdadero$, paso 5), entonces se inicia el proceso de búsqueda (pasos 6 al 19).

En el proceso de búsqueda, $listVar$ es inicializado utilizando el procedimiento FillListVarEval (este procedimiento se explicará más adelante). También es inicializado el contador de retrocesos ($back$) y es seleccionada la primera variable de la lista ($nodeVar \in listVar$) (pasos 6 al 8). A continuación, se ejecuta el bucle principal de procedimiento BLS (pasos del 9 al 19):

[2]Hemos seleccionado el algoritmo de 2-C3OPL como algoritmo de consistencia debido al hecho de que maneja restricciones no-normalizadas eficientemente, pero podríamos seleccionar cualquier otro algoritmo de consistencia que guarde los soportes encontrados.

Algoritmo 41: Procedimiento BLS

```
   Datos: Un CSP, P = ⟨X, D, R⟩
   Resultado: verdadero (hay una solución ⇒ un valor para cada variable) o  falso (no encuentra
              solución)
 1 principio
 2     InitilizeSet(C)
 3     Last ← NIL
 4     continue ← 2C3OPL(P, Last, C)
 5     si (continue = verdadero) entonces
 6         back ← 0
 7         listVar ← FillListVarEval(C)
 8         nodeVar ← primer nodo de listVar
 9         mientras (nodeVar ≠ NIL ∧ continue = verdadero) hacer
10             continue ←SelectValueBL(nodeVar, Last)
11             si continue = falso entonces
12                 nodeVar ← nodeVar.previous
13                 back + +
14             sino
15                 sol ← CheckAsignBL(nodeVar.X_i)
16                 si sol = 1 entonces
17                     nodeVar ← nodeVar.next
18             si nodeVar ≠ NIL entonces
19                 continue ← verdadero
20     retornar continue
21 fin
```

1. BLS intenta asignar un valor a a la variable X_i de $nodeVar$ ($nodeVar.X_i$) utilizando el procedimiento SelectValueBL (Ver Algoritmo 44). Seguidamente, se verifica la asignación o no de un valor $a \in D_i$ a $nodeVar.X_i$. Si SelectValueBL no pudo realizar la asignación de un valor a la variable $nodeVar.X_i$ (paso 10), entonces se realiza un retroceso (backtracking) al $nodeVar.previous$ (pasos del 11 hasta 13); en caso contrario, BLS verifica esta asignación utilizando el procedimiento CheckAsignBL (paso 15).

2. El procedimiento CheckAsignBL verifica que la asignación de $nodeVar.X_i$ sea válida con las asignaciones anteriormente realizadas. Si la asignación $\langle X_i, a\rangle$ es válida entonces es seleccionado el siguiente nodo de $listVar$ ($nodeVar.next$); en caso contrario, permanece en el nodo ($nodeVar.Xi$) para re-ejecutar el procedimiento CheckAsignBL. En este caso, el algoritmo de consistencia (2-C3OPL) es ejecutado para encontrar un nuevo valor para $nodeVar.X_i$. Este nuevo valor $b \in D_i$ (si lo hubiera) será: el siguiente valor $a \in D_i$ en el caso que $nodeVar.father = NIL$ o, el siguiente soporte b consistente con el conjunto de restricciones C_{ij} que comparten $nodeVar.father$ y $nodeVar.X_i$.

De esta manera, el bucle principal del procedimiento BLS, se ejecutará mientras existan nodos pendientes por asignar o, no exista una asignación factible de valores. En el primer caso, se ha obtenido una solución al problema; y en el segundo caso, el problema no tiene solución, debido a que el dominio D_i se ha agotado para la variable $nodeVar.X_i$ y no se ha encontrado una asignación factible.

Algoritmo 42: Procedimiento FillListVarEval

```
   Datos: C: conjunto de bloques de restricciones obtenido de P = ⟨X, D, R⟩.
   Resultado: listVar vector con nodos nodeVar a ser instanciado.
 1 principio
 2     listVar ← NIL
 3     C_ij ← primer conjunto de bloques de restricciones C_ij ∈ C
 4     mientras (C_ij ≠ NIL) hacer
 5         X_i ← C_ij.X_i
 6         X_j ← C_ij.X_j
 7         found_i ← FindInList(X_i, listVar)
 8         found_j ← FindInList(X_j, listVar)
 9         si (found_i = −1) entonces
10             AddnodeVar(listVar, X_i)
11         si (found_j = −1) entonces
12             AddnodeVar(listVar, X_j)
13         AssignFather(nodeVar.X_i, nodeVar.X_j, found_i, found_j, listVar)
14         auxSLR ← siguiente C_ij ∈ C
15     retornar listVar
16 fin
```

El procedimiento FillListVarEval (ver Algoritmo 42) genera los nodos ($nodeVar$) que se almacenan en la lista $listVar$. Hay un único $nodeVar$ para cada variable X_i. Para ello, FillListVarEval utiliza los conjuntos de bloques de restricciones $C_{ij} \in C$ y verifica que las variables X_i y X_j sean referenciadas por un nodo $nodeVar$ de $listVar$ (procedimiento FindInList). El resultado de esta verificación para las variables X_i y X_j, es almacenado respectivamente en las variables enteras $found_i$ y $found_j$. Seguidamente, es ejecutado el procedimiento AssignFather para determinar la *dependencia* de los nodos. Al finalizar su proceso, FillListVarEval retorna $listVar$ actualizada.

FillListVarEval ejecuta el procedimiento AssignFather (ver Algoritmo 43) para determinar la dependencia en la asignación de valores que tendrá cada nodo de $listVar$. La dependencia de asignación está determinado por la posición que cada $nodeVar$ tiene en la lista. El orden de entrada de los nodos es de i a j. Así, el primer nodo de la lista $nodeVar.X_i$ asigna a su padre el valor NIL ($nodeVar.father = NIL$, indica que $nodeVar.X_i$ no tiene dentro de $listVar$ otro nodo anterior a él

Algoritmo 43: Procedimiento AssignFather

Datos: X_i y X_j son las variables del conjunto de restricciones $C_{ij} \in C$ evaluado; $found_i, found_j$ son las variables enteras que indican la existencia o no de las variables $X_i \wedge X_j \in nodeVar\,de\,listVar$; y $listVar$ es una lista ordenada que almacena nodos con las variables $X_i \in X$.
Resultado: $listVar$ con $X_i.father \wedge X_j.father$ actualizados.

```
1  principio
2      node_i ← node.X_i
3      node_j ← node.X_j
4      si (found_i = −1) entonces
5          si (listVar = NIL) entonces
6              node_i.father ← NIL
7          sino
8              si (node_i.father = NIL) entonces
9                  si (found_j = −1) entonces
10                     node_i.father ← NIL
11                 sino
12                     node_i.father ← X_j
13     sino
14         si (found_j > −1 ∧ X_j.father = NIL ∧ found_i < found_j) entonces
15             node_j.father ← X_i
16     si (found_j = −1) entonces
17         si (found_i = −1) entonces
18             node_j.father ← X_i
19         sino
20             si (node_j.father = NIL ∧ found_i > found_j) entonces
21                 node_j.father ← X_i
22     sino
23         si (found_j > −1 ∧ node_i.father = NIL ∧ found_j < found_i) entonces
24             node_i.father ← X_j
25 fin
```

con el que $nodeVar.X_i$ esté relacionado) y el siguiente nodo $nodeVar.X_j$ asigna a su padre el valor $nodeVar.father = X_i$. De esta manera, el padre de un nodo o es NIL o es un nodo anterior con el que comparte un bloque de restricciones C_{ij}, que se encuentra almacenado en la lista $listVar$. En las iteraciones siguientes la asignación del $father$ dependerá de: (a) si sólo una de las variables está actualmente referenciada por un $nodeVar$, entonces ésta será el $father$; (b) si ambas variables están actualmente almacenadas, entonces la primera de ellas será el $father$ (siempre que la otra variable tenga su $node.father = NIL$).

Finalmente, el procedimiento SelectValueBL (ver Algoritmo 44) es el responsable de la instanciación de las variables ($nodeVar.Xi$). A diferencia de otros algoritmos de búsqueda, los valores no son seleccionados a ciegas. Se realizan dos verificaciones

Algoritmo 44: Procedimiento SelectValueBL

```
   Datos: Conjunto C que contiene todos C_ij (Conjuntos de restricciones no-normalizadas obtenidas
          del problema original P = ⟨X, D, R⟩); nodeVar de listVar a ser instanciado
   Resultado: Variable entera result y var.value asignada
 1 principio
 2     result ← 1
 3     var ← nodeVar.X_i
 4     varf ← nodeVar.father
 5     si (var.value = dummyvalue) entonces
 6         si (nodeVar.father = NIL) entonces
 7             var.value ← primer valor de su dominio
 8         sino
 9             var.value ← Last[varf, var, varf.value]
10     sino
11         nextV ← siguiente valor de var.value
12         si (nodeVar.father = NIL) entonces
13             si (nextV ≠ NIL) entonces
14                 var.value ← nextV
15             sino
16                 var.value ← dummyvalue
17                 result ← 0
18         sino
19             si (nextV ≠ NIL) entonces
20                 restric ← FindRestric(nodeVar.father, var, C)
21                 var.value ← nextV
22                 result ←FindNewSuppLast(restric, nodeVar.father, varf.value, var, var.value)
23                 si result = 0 entonces
24                     var.value ← dummyvalue
25             sino
26                 result ← 0
27                 var.value ← dummyvalue
28     retornar result ∧ var.value
29 fin
```

para determinar el valor que será seleccionado: primero, es verificada la instanciación actual de la variable y segundo, es verificada la existencia del padre $nodeVar.father$.

Así, la variable X_i es instanciada a $\langle X_i, a\rangle$ considerando las siguientes reglas:

- Si $X_i = dummyvalue \wedge nodeVar.father = NIL$, entonces el valor a es el primer valor del dominio D_i. (Ver pasos 5 al 7, del Algoritmo 44).
- Si $X_i = dummyvalue \wedge nodeVar.father \neq NIL$, entonces el valor a es el soporte almacenado en $Last$ por su padre. (Ver pasos 5, 8 y 9).
- Si $X_i \neq dummyvalue \wedge nodeVar.father = NIL$, entonces el valor a será el siguiente valor $a' \in D_i$ donde $a' > a$ (pasos 10 al 14) o $a = dummyvalue$ en el caso de que no haya un siguiente valor (ver pasos 10, 15 al 17).

- Si $X_i \neq dummyvalue \wedge nodeVar.father \neq NIL$, entonces el valor a será el nuevo soporte $a' \in D_i \mid a' > a$ consistente con su $father.value$ utilizando el procedimiento FindNewSuppLast (ver pasos del 18 al 24) o $a = dummyvalue$ si no se encontró un soporte (ver pasos 18, 25 al 27).

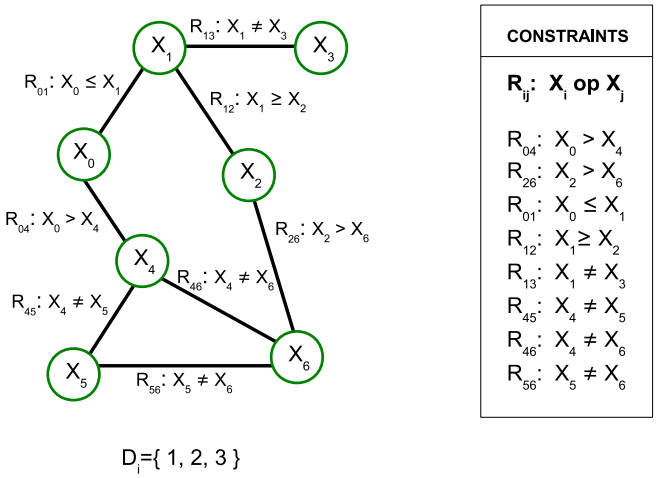


Figura 5.1: Ejemplo de CSP binario-normalizado presentado en [1].

La Figura 5.1 (izquierda), muestra un CSP presentado en [1], que será resuelto utilizando BLS. Las restricciones fueron ordenadas utilizando la heurística de ordenación de restricciones COH [107] (ver Figura 5.1, derecha). Para la resolución de este problema, BLS ejecuta el algoritmo 2-C3OPL que completa la matriz *Last* como se muestra en la Tabla 5.1 y retorna que P' es 2-consistente. Posteriormente, $listVar$ es llenada utilizando el procedimiento FillListVarEval. Este procedimiento determina el orden de asignación de variables y las relaciones de precedencia entre las variables (ver Figura 5.2).

Así, las variables X_0 y X_2 son instanciadas con el primer valor de sus dominios, mientras que las variables X_4, X_6, X_1, X_3 y X_5 son instanciadas en función del valor de sus padres, utilizando para ello los valores almacenados en la matriz *Last*. Se observa que sólo se realiza un único retroceso, debido a que la asignación $\langle X_6, 1\rangle$ obtenida de su padre $\langle X_4, 1\rangle$, no es válida con la asignación $\langle X_2, 2\rangle$ realizada anteriormente. En este caso, SelectValueBL busca un nuevo soporte para X_2 y, con

Tabla 5.1: Valores de la matriz *Last* una vez que el procedimiento 2-C3OPL es ejecutado para el ejemplo mostrado en la Figura 5.1. En este caso, se utilizó: *dummy-value-2*: $d2 = -5$ (como el valor eliminado de D_i) y *dummy-value*: $dv = -1$.

$Last[X_0, X_4, 1] = d2$	$Last[X_0, X_4, 2] = 1$	$Last[X_0, X_4, 3] = 1$
$Last[X_4, X_0, 1] = 3$	$Last[X_0, X_4, 2] = 3$	$Last[X_0, X_4, 3] = d2$
$Last[X_2, X_6, 1] = d2$	$Last[X_2, X_6, 2] = 1$	$Last[X_2, X_6, 3] = 1$
$Last[X_6, X_2, 1] = 3$	$Last[X_6, X_2, 2] = 3$	$Last[X_6, X_2, 3] = d2$
$Last[X_0, X_1, 1] = dv$	$Last[X_0, X_1, 2] = 2$	$Last[X_0, X_1, 3] = 3$
$Last[X_1, X_0, 1] = d2$	$Last[X_1, X_0, 2] = 2$	$Last[X_1, X_0, 3] = 3$
$Last[X_1, X_2, 1] = dv$	$Last[X_1, X_2, 2] = 2$	$Last[X_1, X_2, 3] = 2$
$Last[X_2, X_1, 1] = dv$	$Last[X_2, X_1, 2] = 3$	$Last[X_2, X_1, 3] = 2$
$Last[X_1, X_3, 1] = dv$	$Last[X_1, X_3, 2] = 1$	$Last[X_1, X_3, 3] = 1$
$Last[X_3, X_1, 1] = 3$	$Last[X_3, X_1, 2] = 3$	$Last[X_3, X_1, 3] = 2$
$Last[X_4, X_5, 1] = 2$	$Last[X_4, X_5, 2] = 1$	$Last[X_4, X_5, 3] = dv$
$Last[X_5, X_4, 1] = 2$	$Last[X_5, X_4, 2] = 1$	$Last[X_5, X_4, 3] = 1$
$Last[X_4, X_6, 1] = 2$	$Last[X_4, X_6, 2] = 1$	$Last[X_4, X_6, 3] = dv$
$Last[X_6, X_4, 1] = 2$	$Last[X_6, X_4, 2] = 1$	$Last[X_6, X_4, 3] = dv$
$Last[X_5, X_6, 1] = 2$	$Last[X_5, X_6, 2] = 1$	$Last[X_5, X_6, 3] = 1$
$Last[X_6, X_5, 1] = 3$	$Last[X_6, X_5, 2] = 1$	$Last[X_6, X_5, 3] = dv$

$X_2 = 3$, BLS puede instanciar el resto de las variables y conseguir una solución al problema.

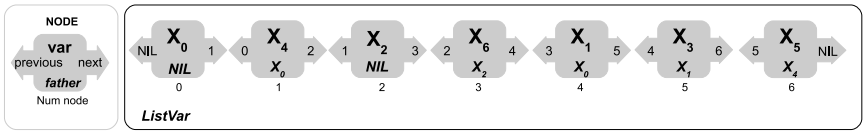


Figura 5.2: Lista *listVar* obtenida al ejecutar el procedimiento FillListVarEval en BLS para el ejemplo de la Figura 5.1.

La Figura 5.3, muestra el proceso de búsqueda realizado por BLS, FC y BT, para el ejemplo de la Figura 5.1. Observe que BLS y BT ejecutan una única vez el algoritmo de consistencia (en la etapa de pre-proceso), pero FC requiere realizar ocho ejecuciones del proceso de consistencia para hacer más eficiente el proceso de búsqueda. Así, en problemas juguete, el proceso de búsqueda es más eficiente con FC y RFLA que con BLS o BT, pero en dominios grandes, cómo veremos en el Capítulo 7, con los problemas de asignación de horarios ferroviarios, FC y RFLA realizan gran consumo de tiempo porque el proceso de consistencia debe repetirse varias veces, y los dominios deben ser almacenadas en caso de que se realice un retroceso. Para

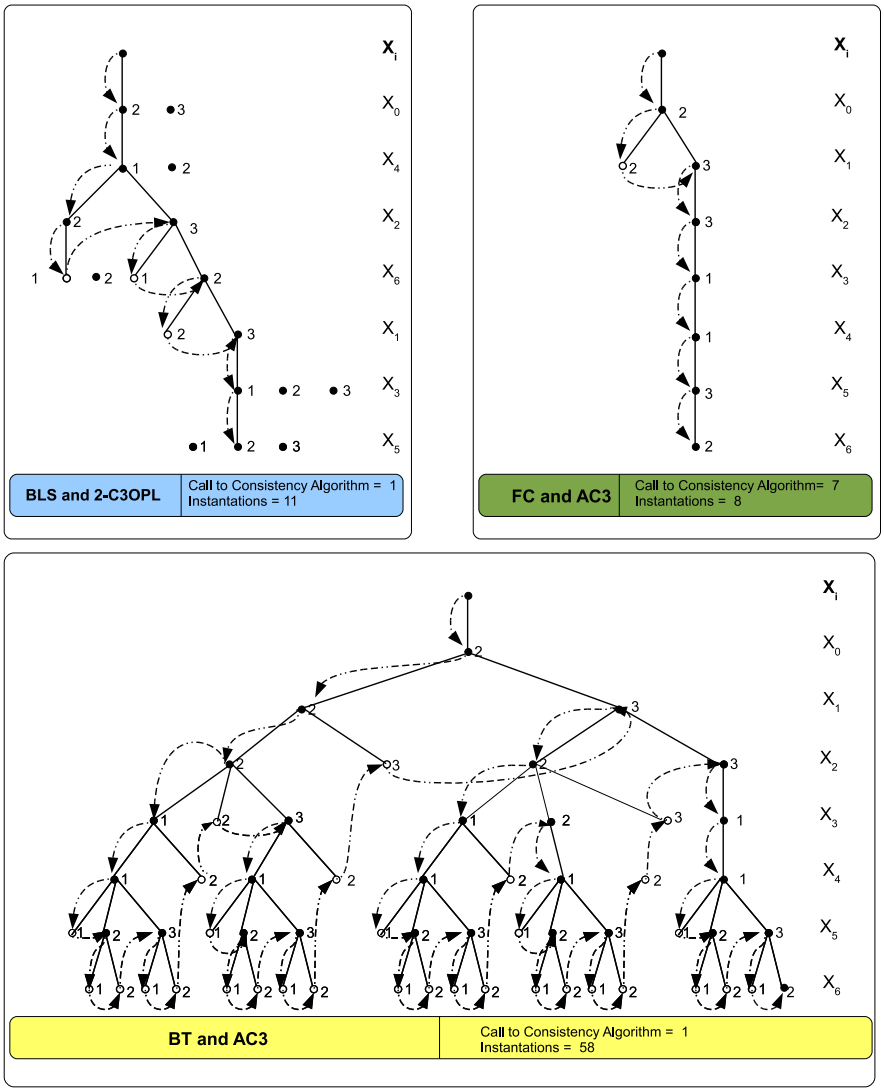


Figura 5.3: Árbol de búsqueda realizado por BLS con 2-C3OPL (arriba a la izquierda), FC con AC3 (arriba a la derecha) y BT con AC3 (abajo) para el CSP de la Figura 5.1.

este tipo de problemas, BLS es voraz y puede encontrar una solución antes que FC y RFLA.

5.3. Algoritmo SchTrains

Los algoritmos independientes del dominio son versátiles debido a que pueden ser utilizados en cualquier contexto con un desempeño aceptable, ya que ellos no explotan características específicas de un determinado dominio. Sin embargo, los algoritmos dependientes del dominio, sí que explotan características de ese dominio en particular, y son más eficientes que los algoritmos independientes del dominio en este tipo de problemas.

De esta forma, utilizamos el conocimiento que tenemos del problema de asignación de horarios ferroviarios para desarrollar un algoritmo de búsqueda para este tipo de problemas llamado: SchTrains.

La información del dominio proporcionada por el usuario es almacenada en un fichero *pp*: número de trenes planificados en cada dirección (T_U y T_D). Para cada tren asignamos un valor $v \in \{d, u\}$ ($t_i \in T_D \Rightarrow v = d$ y $t_i \in T_U \Rightarrow v = u$) y sus correspondientes variables. En *pp* también se indica la clasificación realizada en las restricciones. Esta es realizada en cuatro sub-conjuntos llamados: *precedencia*, *duración*, *capacidad* y *frecuencia*. Por simplicidad, los trenes en T_U son asignados antes que los trenes en T_D, lo cual permite reducir los chequeos de restricciones de capacidad. Adicionalmente, se realiza el proceso de consistencia en etapa de pre-proceso para ayudar al algoritmo SchTrains a reducir el espacio de búsqueda. Dado que el problema modelado es un problema no-normalizado, seleccionamos el algoritmo 2-C3OPL como técnica de consistencia.

El procedimiento SchTrains (ver Algoritmo 45) mantiene estructuras similares a BLS y utiliza las mismas estructuras C y $Last$ explicadas anteriormente (debido a que ambas son requeridas por 2-C3OPL), pero la lista enlazada $listVar$ contiene un $nodeVar$ más simple (sin el campo $father$). Así, la estructura $nodeVar$ en SchTrains está definida por los siguientes campos: var, $next$ y $previous$.

SchTrains trabaja de la siguiente forma: en primer lugar, inicializa las estructuras C y $Last$ (Algoritmo 45, pasos 2 y 3), que serán usadas por 2-C3OPL. Seguidamente, es ejecutado 2-C3OPL (paso 4) para forzar la 2-consistencia y con ello la matriz $Last$

Algoritmo 45: Procedimiento SchTrains

```
   Datos: Un CSP, P = ⟨X, D, R⟩, los parámetros del problema: pp
   Resultado: verdadero (hay solución ⇒ un valor para cada variable) o  falso (no hay solución)
 1 principio
 2     InitilizeSet(C)
 3     Last ← NIL
 4     continue ← 2C3OPL(P, Last, C)
 5     si (continue = verdadero) entonces
 6         back ← 0
 7         listVar ← FillListVarTrain(pp)
 8         nodeVar ← primer nodo de listVar
 9         mientras (nodeVar ≠ NIL ∧ continue = verdadero) hacer
10             continue ← SelecValueTrain(nodeVar)
11             si continue = falso entonces
12                 nodeVar ← nodeVar.previous
13                 back + +
14             sino
15                 sol ← CheckAsignTrain(nodeVar.X_i, listVar, pp)
16                 si (sol = verdadero) entonces
17                     nodeVar ← nodeVar.next
18             si nodeVar ≠ NIL entonces
19                 continue ← verdadero
20     retornar continue
21 fin
```

es actualizada con los soportes encontrados por el proceso de 2-consistencia. Si el problema resultante P' es consistente, entonces se inicia el proceso de búsqueda (paso 5).

El proceso de búsqueda consiste básicamente de dos sub-procesos inicialización (Algoritmo 45, pasos del 6 al 8) y búsqueda (pasos del 9 al 19). En el subproceso de Inicialización, $listVar$ es inicializado por el procedimiento FillListVarTrain. Este procedimiento almacena las variables en el siguiente orden: en primer lugar, es almacenada la variable $S00$ (tiempo de inicio -*start time*-); luego son almacenadas todas las variables concernientes a los trenes en T_D; posteriormente son almacenadas las variables concernientes a los trenes de T_U y finalmente, es almacenada la variable $E01$ (tiempo final -*end time* o *makespan*-).

Cuando ha finalizado el sub-proceso de inicialización, es ejecutado el sub-proceso de búsqueda. La búsqueda consiste en un bucle que se inicia seleccionando el primer nodo $nodeVar$ de $listVar$. Después de esto, SchTrains intenta asignar un valor a a la variable X_i del $nodeVar$ ($\langle X_i, a\rangle$) utilizando el procedimiento SelectValueTrain (paso 10). El procedimiento SelectValueTrain verifica si la variable X_i ha tenido previamente un valor asignado. De ser así, X_i es asignada con el siguiente valor

de su dominio; en caso contrario, es asignada con el primer valor de su dominio. Seguidamente, se verifica la asignación o no de un valor $a \in D_i$ a $nodeVar.X_i$. Si SelectValueTrain no pudo realizar una asignación de valor a la variable $nodeVar.X_i$, entonces se realiza un retroceso (backtracking) a $nodeVar.previous$ (ver Algoritmo 45, pasos del 11 al 13). En caso contrario, SchTrain verifica esta asignación utilizando el procedimiento CheckAsignTrain (paso 15).

Si esta asignación $\langle X_i, a \rangle$ es válida, entonces selecciona el próximo nodo de $listVar$ ($nodeVar.next$, pasos 16 y 17); de lo contrario, permanece en este nodo ($nodeVar.Xi$) (pasos 18-19). En este último caso el procedimiento SelecValueTrain busca un nuevo valor k para $nodeVar.X_i$; donde $k > a$ y $k \in D_i$. Si encuentra un nuevo valor k para $nodeVar.X_i$, entonces es ejecutado nuevamente CheckAsignTrain con $\langle X_i, k \rangle$. De esta forma, el ciclo principal del procedimiento SchTrains (pasos del 9 al 19) se ejecutará mientras que existan nodos pendientes de asignación o no exista una asignación de valores factible. En el primer caso, se ha obtenido una solución al problema, y en el segundo caso, el problema no tiene solución porque el dominio D_i de la variable $nodeVar.X_i$ se ha agotado, sin que se haya encontrado una asignación de valores factible.

El procedimiento CheckAsignTrain (ver Algoritmo 46) comienza inicializando la variable Booleana $result$ a $verdadero$ y la variable Booleana $found$ a $falso$ (paso 2). El procedimiento CheckAsignTrain realiza el proceso de chequeo en forma selectiva, dependiendo de la variable $nodeVar.X_i$ a ser verificada:

- Si $nodeVar.X_i$ es igual a $S00$, entonces no se realizan chequeos porque $S00$ es una variable ficticia que indica el inicio de tiempo (*start time*) del problema y el *conocimiento del dominio* indica que la variable X_j no está instanciada todavía (ver Algoritmo 46, pasos 3 y 4).

- Si $nodeVar.X_i$ es igual a $E01$, entonces únicamente es chequeado el subconjunto de restricciones de *precedencia*. La asignación inicial para la variable $E01$ es fijada al valor más grande asignado de aquellas variables que comparten restricciones con la variable $E01$ (ver Algoritmo 46, pasos 5 al 11).

- Si $nodeVar.X_i \in T$, entonces hay cuatro casos (ver Algoritmo 46, paso 12):

Algoritmo 46: Procedimiento CheckAsignTrain

Datos: $nodeVar.X_i$, $listVar$ y los sub-conjuntos de restricciones de *pp*: *precedencia*, *duración*, *cruce* y *frecuencia*.
Resultado: Variable Booleana $result = verdadero$ cuando $\langle X_i, a\rangle$ es válida o $result = falso \wedge \langle X_i, a\rangle$ actualizada

```
1  principio
2      result ← verdadero, found ← falso
3      si (nodeVar.X_i = S00) entonces
4          /* no se realiza chequeo */
5      si (nodeVar.X_i = E01) entonces
6          /* Únicamente son evaluadas las restricciones de precedencia donde se referencia a E01 */
7          R_ij ← primera restricción del subconjunto de restricciones de precedencia
8          mientras (R_ij ≠ NIL) hacer
9              si (nodeVar.X_i ∈ R_ij) ∧ a < b | a ∈ ⟨X_i, a⟩; b ∈ ⟨X_j, b⟩ entonces
10                 cambia la instanciación ⟨X_i, a⟩ por ⟨X_i, b⟩
11             R_ij ← siguiente restricción
12     si (nodeVar.X_i ∈ T) entonces
13         si (nodeVar.X_i ∈ T_U) entonces
14             /* Únicamente son evaluadas las restricciones de cruce donde se referencia a
                  nodeVar.X_i */
15             cSS ← subconjunto de restricciones de cruce
16             value = dummyvalue
17             result ← CheckSet(all = verdadero, cSS, nodeVar.X_i, listVar, NIL, value)
18             si (result = falso) entonces
19                 cambia la instanciación ⟨X_i, a⟩ por ⟨X_i, value⟩
20         si (result = verdadero) entonces
21             /* Únicamente son evaluadas las restricciones de frecuencia donde se referencia a
                  nodeVar.X_i */
22             cSS ← subconjunto de restricciones de frecuencia
23             result ← CheckSet(all = verdadero, cSS, nodeVar.X_i, listVar, NIL, NIL)
24         si (result = verdadero) entonces
25             /* Únicamente son evaluadas las restricciones de precedencia donde se referencia a
                  nodeVar.X_i */
26             cSS ← subconjunto de restricciones de precedencia
27             found ← falso
28             result ← CheckSet(all = falso, cSS, nodeVar.X_i, listVar, found, NIL)
29         si (found = falso ∧ result = verdadero) entonces
30             /* Únicamente son evaluadas las restricciones de duración donde se referencia a
                  nodeVar.X_i */
31             cSS ← subconjunto de restricciones de duración
32             result ←CheckSet(all = falso, cSS, nodeVar.X_i, listVar, NIL, NIL)
33     retornar result
34 fin
```

Algoritmo 47: Procedimiento CheckSet

Datos: Variable Booleana *all*, subconjuntos de restricciones cSS, $nodeVar.X_i$, $listVar$, variable Booleana $found$ y variable entera $value$.
Resultado: Variable Booleana $response = verdadero$ cuando $\langle X_i, a\rangle$ es válida o $response = falso$ cuando $\langle X_i, a\rangle$ no es válida; y las variables $found \wedge value$ actualizadas

```
1  principio
2      response ← verdadero
3      R_ij ← primera restricción del subconjunto de restricciones cSS
4      si (all = verdadero) entonces
5          /*Únicamente son evaluadas las restricciones en cSS donde se referencia a nodeVar.X_i */
6          mientras (R_ij ≠ NIL ∧ result = verdadero) hacer
7              si ((nodeVar.X_i ∈ R_ij) ∧ ((⟨X_i, a⟩, ⟨X_j, b⟩) ∉ R_ij)) entonces
8                  value ← b /* únicamente cuando su valor de entrada es diferente a NIL */
9                  response ← falso
10             sino
11                 R_ij ← próxima restricción
12     sino
13         /*Únicamente son evaluadas las restricciones en cSS donde se referencia a nodeVar.X_i */
14         mientras (R_ij ≠ NIL ∧ found ≠ verdadero) hacer
15             si (nodeVar.X_i ∈ R_ij) entonces
16                 found ← verdadero /* únicamente cuando su valor de entrada es diferente a
                   NIL */
17                 si ((⟨X_i, a⟩, ⟨X_j, b⟩) ∉ R_ij) entonces
18                     response ← falso
19             sino
20                 R_ij ← próxima restricción
21     retornar response
22 fin
```

1. Si $nodeVar.X_i$ pertenece a T_U, entonces se verifica el subconjunto de restricciones de *cruce* (ver Algoritmo 46, paso 13) utilizando el procedimiento CheckSet (ver Algoritmo 47, pasos del 2 al 11). Como las restricciones de *precedencia* hacen con $nodeVar.X_i$ igual a $E01$, si una restricción de *cruce* no es satisfecha, entonces $result$ es cambiada a $falso$ y la instanciación actual de $\langle X_i, a\rangle$ es cambiada a $\langle X_i, value\rangle$; donde $value = b$; $b \in D_j$; $X_j \in nodeVar$ previamente instanciado ($\langle X_j, b\rangle$); X_i, X_j son las variables en R_{ij} y $R_{ij} \in$ subconjunto de restricciones de *cruce*. Así, la instanciación de $nodeVar.X_i$ es rápidamente aproximada a los valores correctos.
2. Si $result$ igual a $verdadero$, entonces cada R_{ij} perteneciente a las restricciones de *frecuencia* donde es referenciado $nodeVar.X_i$, es evaluada (ver Algoritmo 46, paso 20). Si una R_{ij} no es satisfecha, entonces la variable $result$ es cambiada a $falso$ (paso 23).
3. Si el chequeo anterior fue válido ($result = verdadero$), entonces las R_{ij} pertenecientes a las restricciones de *precedencia* donde es referenciada $nodeVar.X_i$, son chequeadas (ver Algoritmo 46, paso 24) utilizando el procedimiento CheckSet (ver Algoritmo 47, pasos 2, 3 y del 12 al 20). Si R_{ij} es encontrado, entonces la variable $found$ es cambiada a $verdadero$ y es chequeada la R_{ij}. Si R_{ij} no se satisface, entonces la variable $result$ es cambiada a $falso$. Hay que destacar que cuando $found$ es igual a $verdadero$, esto es utilizado para evitar el chequeo del subconjunto de restricciones de *duración* porque el *conocimiento del dominio* indica que la variable X_j no está todavía instanciada.
4. Si $result$ es igual a $verdadero$ y $found$ es igual a $falso$, entonces la R_{ij} perteneciente a las restricciones de *duración* donde es referenciada $nodeVar.X_i$, es evaluada (ver Algoritmo 46, paso 29) utilizando el procedimiento CheckSet (ver Algoritmo 47, pasos 2, 3 y del 12 hasta el 20). Si es encontrado R_{ij}, entonces la variable $found$ es cambiada a $verdadero$ (para salir del ciclo) y es chequeada R_{ij}. Si R_{ij} no es satisfecha, entonces la variable $result$ es cambiada a $falso$.

El procedimiento CheckSet (ver Algoritmo 47) verifica que la instanciación actual

$\langle Xi, a\rangle$ del $nodeVar.X_i$ con las instanciaciones previas $\langle Xj, b\rangle$ sean satisfechas en el subconjunto de restricciones. Esto es realizado en dos vías:

1. para todas las restricciones en el conjunto (ver Algoritmo 47, pasos 2 al 11), cuando la variable Booleana de entrada *all* es igual a *verdadero* (esta vía es llamada en el Algoritmo 46, pasos 17 y 22), y

2. para una única restricción (si esta restricción existe) (ver Algoritmo 47, pasos 12 al 20), cuando la variable *all* es igual a $falso$ (esta vía es llamada en el Algoritmo 46, pasos 28 y 32).

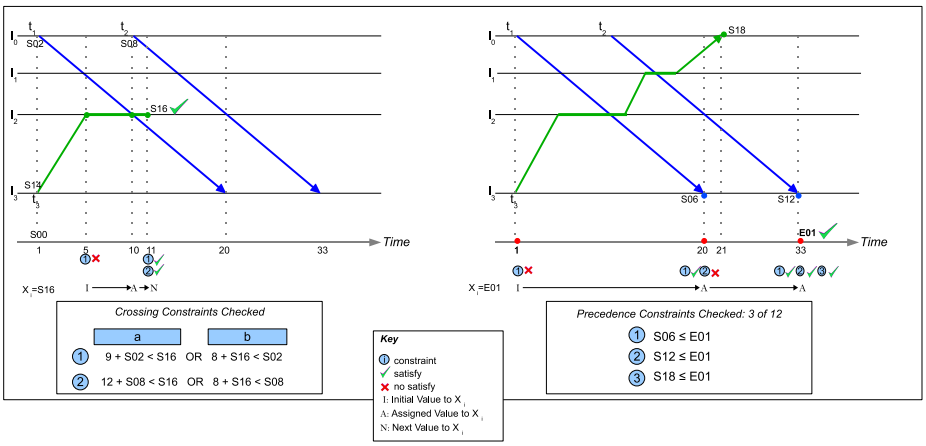


Figura 5.4: Evaluación de las restricciones de *cruce* $\Rightarrow nodeVar.X_i \in T_U$ (izquierda) y de las restricciones de *precedencia* cuando $nodeVar.X_i = E01$ (derecha) .

La Figura 5.4 muestra dos ejemplos donde las restricciones de *cruce* (izquierda) y las restricciones de *precedencia* (derecha) son evaluadas para dos $nodeVar.X_i$ diferentes. La Figura 5.4 (a la izquierda) muestra como $nodeVar.X_i$ igual a $S16$ donde es evaluada la variable $S16 \in T_U$. Primero es chequeado el valor inicial de $S16$ (paso 15 del Algoritmo SchTrains), ejecutando los pasos del 2 al 11 del procedimiento CheckSet. Segundo, debido a que la condición en el paso 7 del Algoritmo 47 es *verdadera*, es decir, $R_{ij} : 9 + S02 < S16 \vee 8 + S16 < S02$ no se satisface para $\langle S16, 5\rangle$, entonces el resultado de resolver $9 + S02 = 9 + 1 = 10$ (donde el valor 1 se obtiene de la instanciación de $\langle S02, 1\rangle$) es retornado por CheckSet en la variable entera *value* y *result* cambia a $falso$. La variable *value* es usada para actualizar la instanciación de $S16$ a $\langle S16, 10\rangle$. Tercero, dado que $nodeVar \neq NIL$, entonces

la variable Booleana *continue* es cambiada a *verdadero* (ver Algoritmo 45, pasos 18 y 19) y se realiza un nuevo ciclo en el procedimiento SchTrains. Finalmente, con esta nueva instanciación ($\langle S16, 10\rangle$), el procedimiento SelectValueTrain asigna el siguiente valor en D_i, por lo que la instanciación es cambiada a $\langle S16, 11\rangle$ y son nuevamente ejecutados los pasos del 2 al 11 del procedimiento CheckSet. En este caso *result* es igual a *verdadero*. Las instanciaciones para $\langle S16, t\rangle$ con $t = \{6, 7, 8, 9, 10\}$ no son verificadas porque se ha utilizado el *conocimiento del dominio*, donde se *conoce* que esas instanciaciones no son válidas y que *pueden evitarse*. De esta forma ahorramos instanciaciones, chequeos y tiempo de cómputo.

La Figura 5.4 (en la derecha) muestra un ejemplo de como es evaluado el $nodeVar.X_i$ igual a $E01$. En este caso, es ejecutado el paso 5 del Algoritmo 46. En este caso, las restricciones de *precedencia* evaluadas cambian la instanciación de $E01$ dos veces: la primera de $\langle E01, 1\rangle$ a $\langle E01, 20\rangle$ porque la restricción de precedencia $R_{ij} : S06 \leq E01$ no es satisfecha y posteriormente de $\langle E01, 20\rangle$ a $\langle E01, 33\rangle$ porque la restricción de precedencia $R_{ij} : S12 \leq E01$ no es satisfecha. Los nuevos valores para $E01$ son obtenidos de las instanciaciones de las otras variables $X_j \in R_{ij}$, es decir, $\langle S06, 20\rangle$ y $\langle S12, 33\rangle$, respectivamente. De esta forma son evitadas las instanciaciones $\langle E01, t\rangle$ con $t = \{2..19, 21..32\}$ y sus chequeos y, $E01$ obtiene una instanciación optima.

5.4. Conclusiones

En este capítulo hemos presentado dos algoritmos de búsqueda: BLS y SchTrains. El algoritmo BLS combina la simplicidad de los algoritmos con enfoque loop-back con la matriz *Last* obtenida del proceso de consistencia aplicado en la etapa de pre-proceso. Adicionalmente, BLS utiliza las restricciones para decidir el orden de asignación de las variables. En la búsqueda, el algoritmo BLS tuvo un mejor comportamiento que los algoritmos RFLA, FC y BT, en las instancias del problema de asignación de horarios ferroviarios. Además, el tiempo de cómputo fue reducido en un 80 % en relación a RFLA y, dicho tiempo fue reducido en un 99 % en relación a FC y BT en instancias de este problema con solución. El Algoritmo BLS no pudo encontrar una solución en un 12 % de los problemas evaluados. Sin embargo, esta técnica de búsqueda propuesta sigue siendo útil para encontrar una solución en

problemas con una talla grande de dominio.

SchTrains es un algoritmo de búsqueda dependiente del dominio de asignación de horarios ferroviarios, que combinado con una técnica de consistencia que almacene soportes, puede encontrar muy rápidamente una solución al problema. El algoritmo SchTrains tuvo mejor desempeño en todas las instancias que los algoritmos RFLA, FC, BT y BLS. Dicha mejora estuvo por encima de un orden de magnitud.

Capítulo 6

Resultados experimentales

Introducción

En este capítulo, comparamos empíricamente el comportamiento de los algoritmos de arco-consistencia, 2-consistencia y de búsqueda propuestos, frente a los algoritmos de arco-consistencia y de búsqueda presentes en la literatura.

La determinación de qué algoritmo es superior a los demás sigue siendo difícil de evaluar. Los algoritmos a menudo se han comparado mediante la observación de su desempeño (tiempo de cómputo, número de chequeos, número de podas, etc.) en problemas de referencia (benchmark problems) y en problemas aleatorios a partir de una distribución sencilla y uniforme. Por una parte, la ventaja de utilizar los problemas de referencia es que si son problemas interesantes (para alguien), entonces la información sobre el algoritmo que funciona bien en estos problemas también es interesante. Sin embargo, a pesar de que un algoritmo supere a cualquier otro cuando es aplicado a un problema de referencia concreto, es difícil extrapolar esta característica a los problemas generales. Por otra parte, una ventaja de utilizar los problemas aleatorios es que hay muchos de ellos, y los investigadores pueden diseñar experimentos cuidadosamente controlados. Sin embargo, un inconveniente de los problemas aleatorios es que no reflejan en muchas ocasiones situaciones de la vida real.

Para la evaluación se generaron problemas aleatorios (normalizados y no-normalizados) y se utilizaron problemas benchmarks presentes en la literatura. Para medir el rendimiento de los algoritmos se consideraron las siguientes medidas de eficiencia:

- Para los algoritmos de arco-consistencia: tiempo de cómputo, número de podas, número de chequeos de restricciones, número de propagaciones y número de soportes encontrados.
- Para los algoritmos de 2-consistencia: tiempo de cómputo, número de podas, número de conjuntos de restricciones procesados, número de chequeos de restricciones, número de propagaciones y número de soportes encontrados.
- Para los algoritmos de búsqueda: tiempo de cómputo en encontrar la primera solución (si existe).

Todos los algoritmos evaluados fueron implementados en C. Los experimentos fueron realizados en un ordenador con procesador Intel Core 2 Quad (de 2.83 GHz velocidad del procesador y 3 GB RAM).

6.1. Problemas aleatorios

En esta sección se trabajará con problemas binarios aleatorios. Los problemas pueden ser consistentes o inconsistentes, los problemas consistentes pueden tener o no solución. Adicionalmente, en los problemas pueden variar: el número de variables, la talla del dominio y el número de restricciones. Las restricciones de los problemas pueden estar equilibradas o no-equilibradas y pueden ser normalizadas o no-normalizadas. Las restricciones no-normalizadas pueden variar en el número de restricciones (2, 3 o más restricciones) entre el mismo par de variables. Finalmente, el operador de las restricciones puede ser sólo de orden $\{<, >\}$ o una combinación de operadores aritméticos $\{<, \leq, >, \geq, =, \neq\}$.

6.1.1. Problemas aleatorios normalizados

Evaluación de problemas con restricciones de orden

Una instancia de CSP aleatorio normalizado con restricción de orden está caracterizada por la 3-tupla $\langle n, d, m\rangle$, donde n es el número de variables, d la talla del dominio y m el número de restricciones binarias. En cada instancia fijamos dos parámetros y variamos el otro para evaluar el comportamiento de los algoritmos

cuando este parámetro se incrementa. Las restricciones fueron generadas de dos formas equilibradas, donde se distribuyen uniformemente las restricciones entre las variables y no equilibradas, donde hay variables que participan más que otras en las restricciones. Todas las restricciones son binarias, normalizadas y fueron generadas de la siguiente forma: $a+X_i < b+X_j$ con a y $b \in N$. En todos los problemas las variables mantienen la misma talla de dominio y todos los problemas son consistentes. Se evaluaron 50 casos de pruebas para cada instancia del problema.

Para este tipo de problemas se comparó el desempeño de los algoritmos propuestos: AC3-OP y AC2001-OP en relación a sus algoritmos base: AC3 y AC2001/3.1, respectivamente, debido a que los mecanismos de propagación de los algoritmos propuestos están diseñados para este tipo de restricciones.

- **Evaluación de problemas equilibrados con respecto al número de variables**

La Tabla 6.1 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq) y el número de propagaciones (prop) utilizando los algoritmos de arco-consistencia: AC3, AC3-OP, AC2001/3.1 y AC2001-OP, en problemas binarios, normalizados aleatorios, equilibrados con restricciones de orden, donde el número de variables n fue incrementada de 100 a 800, la talla de dominio d fue fijada en 100 y el número de restricciones m fue fijado en 500: $\langle n, 100, 500\rangle$.

Los resultados muestran que todos los algoritmos realizaron la misma cantidad de podas. Los algoritmos propuestos AC3-OP y AC2001-OP tuvieron mejor desempeño que sus algoritmos base: AC3 y AC2001/3.1, respectivamente, llegando a reducir a la mitad el número de propagaciones. Realizando una comparación general entre todos los algoritmos (AC3, AC3-OP, AC2001/3.1 y AC2001-OP) en cuanto al número de chequeos y número de propagaciones, el algoritmo AC2001-OP obtuvo mejor desempeño que el resto de algoritmos AC2001/3.1, AC3 y AC3-OP. En cuanto al tiempo de cómputo, en las instancias hasta $n = 200$, el algoritmo AC2001-OP tuvo mejor desempeño que AC2001/3.1, AC3 y AC3-OP, ya que hasta 200 variables, la estructura de almacenamiento de soportes y el hecho de no realizar propagaciones ineficientes, ayuda a AC2001-OP a ser más eficiente; pero a partir de 300 o más variables no fue así, con lo que AC3-OP tuvo mejor desempeño en tiempo que el resto de algoritmos, ya que al no procesar restricciones inversas y

Tabla 6.1: Resultados de las técnicas de arco-consistencia AC3, AC3-OP, AC2001/3.1 y AC2001-OP en problemas binarios, normalizados, equilibrados con restricciones de orden: $\langle n, 100, 500\rangle$.

n	**podas**		**AC3**	**AC3-OP**	**AC2001/3.1**	**AC2001-OP**
100	1.37×10^{03}	*tiempo*	193.72	161.64	96.68	75.62
		cheq	1.18×10^{7}	1.07×10^{7}	2.49×10^{6}	2.49×10^{6}
		prop	5.93×10^{3}	2.10×10^{03}	5.93×10^{3}	2.10×10^{3}
200	1.33×10^{3}	*tiempo*	129.72	114.06	118.74	108.42
		cheq	7.50×10^{6}	6.84×10^{6}	2.61×10^{6}	2.61×10^{6}
		prop	2.53×10^{3}	9.36×10^{2}	2.53×10^{3}	9.36×10^{2}
300	1.23×10^{3}	*tiempo*	96.88	88.78	150.7	148.14
		cheq	5.85×10^{6}	5.34×10^{6}	2.65×10^{6}	2.65×10^{6}
		prop	1.38×10^{3}	5.64×10^{2}	1.38×10^{3}	5.64×10^{2}
400	1.10×10^{3}	*tiempo*	78.14	64.02	204.72	199.08
		cheq	4.72×10^{6}	4.32×10^{6}	2.67×10^{6}	2.67×10^{6}
		prop	8.16×10^{2}	3.39×10^{2}	8.16×10^{2}	3.39×10^{2}
500	1.10×10^{3}	*tiempo*	77.52	74.68	299.12	293.16
		cheq	4.31×10^{6}	4.00×10^{6}	2.68×10^{6}	2.68×10^{6}
		prop	5.26×10^{2}	2.65×10^{2}	5.26×10^{2}	2.65×10^{2}
600	1.07×10^{3}	*tiempo*	63.74	62.5	385.56	379.8
		cheq	4.03×10^{6}	3.77×10^{6}	2.69×10^{6}	2.69×10^{6}
		prop	4.14×10^{2}	2.18×10^{2}	4.14×10^{2}	2.18×10^{2}
700	1.05×10^{3}	*tiempo*	72.88	62.84	513.88	509.34
		cheq	3.83×10^{6}	3.61×10^{6}	2.69×10^{6}	2.69×10^{6}
		prop	3.35×10^{2}	1.84×10^{2}	3.35×10^{2}	1.84×10^{2}
800	1.05×10^{3}	*tiempo*	62.48	57.84	641.28	631.48
		cheq	3.68×10^{6}	3.50×10^{6}	2.70×10^{6}	2.70×10^{6}
		prop	2.80×10^{2}	1.61×10^{2}	2.80×10^{2}	1.61×10^{2}

no requerir de estructuras de almacenamiento de soportes, puede realizar más rápidamente el proceso de chequeo. La Tabla 6.1 también muestra que el número de podas y chequeos de restricciones fueron reducidos cuando el número de variables se incrementó. Esto es debido a que cuando el número de variables se incrementa, la restringibilidad de los problemas decrece debido a que el número de restricciones permanece constante.

- **Evaluación de problemas equilibrados con respecto al número de restricciones**

La Tabla 6.2 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq) y el número de propagaciones (prop) utilizando

Tabla 6.2: Resultados de las técnicas de arco-consistencia AC3, AC3-OP, AC2001/3.1 y AC2001-OP en problemas binarios, normalizados, equilibrados con restricciones de orden: $\langle 100, 100, m \rangle$

m	**podas**		**AC3**	**AC3-OP**	**AC2001/3.1**	**AC2001-OP**
100	2.12×10^2	*tiempo*	12.88	6.32	16.52	15.76
		cheq	8.53×10^5	7.86×10^5	5.36×10^5	5.36×10^5
		prop	1.03×10^2	5.06×10^1	1.03×10^2	5.06×10^1
200	5.25×10^2	*tiempo*	37.76	31.1	31.26	31
		cheq	2.72×10^6	2.50×10^6	1.05×10^6	1.05×10^6
		prop	7.54×10^2	3.09×10^2	7.54×10^2	3.09×10^2
300	8.56×10^2	*tiempo*	80.04	68.98	46.88	43.88
		cheq	5.23×10^6	4.79×10^6	1.55×10^6	1.55×10^6
		prop	2.00×10^3	7.46×10^2	2.00×10^3	7.46×10^2
400	1.13×10^3	*tiempo*	131.94	113.12	67.56	58.7
		cheq	8.30×10^6	7.56×10^6	2.02×10^6	2.02×10^6
		prop	3.74×10^3	1.35×10^3	3.74×10^3	1.35×10^3
500	1.40×10^3	*tiempo*	202.54	167.5	98.5	76.66
		cheq	1.22×10^7	1.11×10^7	2.50×10^6	2.50×10^6
		prop	6.18×10^3	2.23×10^3	6.18×10^3	2.23×10^3
600	1.67×10^3	*tiempo*	284.02	223.1	135.08	94
		cheq	1.63×10^7	1.46×10^7	2.93×10^6	2.93×10^6
		prop	8.92×10^3	3.19×10^3	8.92×10^3	3.19×10^3
700	1.96×10^3	*tiempo*	365.82	273.82	183.02	113.74
		cheq	1.99×10^7	1.76×10^7	3.34×10^6	3.34×10^6
		prop	1.20×10^4	4.19×10^3	1.20×10^4	4.19×10^3
800	2.35×10^3	*tiempo*	534.38	369.4	273.14	145.96
		cheq	2.69×10^7	2.34×10^7	3.77×10^6	3.77×10^6
		prop	1.76×10^4	6.25×10^3	1.76×10^4	6.25×10^3

los algoritmos de arco-consistencia: AC3, AC3-OP, AC2001/3.1 y AC2001-OP, en problemas binarios, normalizados aleatorios, equilibrados con restricciones de orden; donde el número de variables n fue fijado en 100, la talla de dominio d fue fijada en 100 y el número de restricciones fue incrementado de 100 a 800: $\langle 100, 100, m \rangle$.

Al igual que en los problemas anteriores, los algoritmos propuestos AC3-OP y AC2001-OP, tuvieron un mejor desempeño que sus algoritmos base. En la comparación general con el resto de algoritmos, AC2001-OP fue el que tuvo mejor desempeño, debido a que el número de variables es reducido y constante, lo que no incrementa la estructura de almacenamiento de soportes. Así mismo, la Tabla 6.2 muestra que el número de podas y chequeos de restricciones fue aumentando cuando el número de restricciones se incrementó. Esto es debido a que cuando el número de restricciones

se incrementa, la restringibilidad de los problemas también se incrementa debido a que el número de variables permanece constante.

- **Evaluación de problemas no-equilibrados con respecto al número de variables**

La Tabla 6.3 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq) y el número de propagaciones (prop) utilizando los algoritmos de arco-consistencia: AC3, AC3-OP, AC2001/3.1 y AC2001-OP, en problemas binarios, normalizados aleatorios, no-equilibrados con restricciones de orden; donde el número de variables n fue incrementado de 50 a 750, la talla de dominio d fue fijada en 50 y el número de restricciones m fue fijado en 700: $\langle n, 50, 700\rangle$.

Los resultados muestran que todos los algoritmos realizaron la misma cantidad de podas. Los algoritmos propuestos AC3-OP y AC2001-OP tuvieron mejor desempeño que sus algoritmos base: AC3 y AC2001/3.1, respectivamente. Por ejemplo, para $n = 250$, AC3-OP redujo en un 49 % el tiempo de cómputo, también redujo en un orden de magnitud el número de chequeos y en dos órdenes de magnitud el número de propagaciones, en relación a AC3; mientras que AC2001-OP redujo el tiempo de cómputo en un 1.15 %, el número de chequeos en un 1 % y el número de propagaciones en un orden de magnitud. En este tipo de problemas AC3-OP tuvo mejor tiempo de cómputo promedio, mientras que AC2001-OP tuvo el menor número de chequeos y el menor número de propagaciones.

- **Evaluación de problemas no-equilibrados con respecto al número de restricciones**

La Tabla 6.4 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq) y el número de propagaciones (prop) utilizando los algoritmos de arco-consistencia: AC3, AC3-OP, AC2001/3.1 y AC2001-OP, en problemas binarios, normalizados aleatorios, no-equilibrados con restricciones de orden; donde el número de restricciones m fue incrementado de 100 a 800, mientras que el número de variables y la talla de dominio fueron fijadas ambas en 50: $\langle 50, 50, m\rangle$. Los resultados fueron similares a los mostrados en la Tabla 6.2, en cuanto a que AC2001-OP tuvo el mejor desempeño, pero los tiempos de cómputo de AC3-OP y AC2001-OP fueron bastante similares entre sí. Por lo que en este tipo

Tabla 6.3: Resultados de las técnicas de arco-consistencia AC3, AC3-OP, AC2001/3.1 y AC2001-OP en problemas binarios, normalizados y no-equilibrados: $\langle n, 50, 700\rangle$.

n	**podas**		**AC3**	**AC3-OP**	**AC2001/3.1**	**AC2001-OP**
50	8.71×10^2	*tiempo*	93.66	31.22	78.44	31.26
		cheq	1.96×10^6	1.32×10^6	7.52×10^5	7.48×10^5
		prop	8.01×10^3	9.69×10^2	8.01×10^3	9.69×10^2
150	7.18×10^2	*tiempo*	46.88	30.7	61.6	46.58
		cheq	1.59×10^6	9.51×10^5	9.30×10^5	9.29×10^5
		prop	2.03×10^3	2.00×10^1	2.03×10^3	2.00×10^1
250	7.03×10^2	*tiempo*	31.26	15.62	78.4	77.5
		cheq	1.51×10^6	9.68×10^5	9.64×10^5	9.63×10^5
		prop	1.18×10^3	3.96×10^0	1.18×10^3	3.96×10^0
350	7.04×10^2	*tiempo*	38.06	22.54	129.06	110.2
		cheq	1.43×10^6	9.83×10^5	9.79×10^5	9.78×10^5
		prop	1.33×10^3	3.82×10^0	1.33×10^3	3.82×10^0
450	7.06×10^2	*tiempo*	31.26	31.34	180.4	173.08
		cheq	1.31×10^6	9.86×10^5	9.85×10^5	9.84×10^5
		prop	6.97×10^2	1.00×10^0	6.97×10^2	1.00×10^0
550	7.01×10^2	*tiempo*	30.94	15.9	242.76	236.88
		cheq	1.19×10^6	9.91×10^5	9.90×10^5	9.89×10^5
		prop	6.85×10^2	9.80×10^{-1}	6.85×10^2	9.80×10^{-1}
650	7.03×10^2	*tiempo*	31.72	31.2	328.14	321.22
		cheq	1.09×10^6	9.96×10^5	9.95×10^5	9.95×10^5
		prop	6.85×10^2	9.80×10^{-1}	6.85×10^2	9.80×10^{-1}
750	7.02×10^2	*tiempo*	19.96	15.68	411.58	402.58
		cheq	1.02×10^6	9.98×10^5	9.97×10^5	9.97×10^5
		prop	2.79×10^2	4.00×10^{-1}	2.79×10^2	4.00×10^{-1}

de problemas, se obtienen resultados similares con cualquiera de las dos técnicas AC3-OP o AC2001-OP.

Evaluación de problemas con restricciones aritméticas

Una instancia de CSP aleatorio, normalizado con restricciones aritméticas está caracterizada por la 3-tupla $\langle n, d, m\rangle$, donde n es el número de variables, d la talla del dominio y m el número de restricciones binarias. En cada instancia fijamos dos parámetros y variamos el otro para evaluar el desempeño de los algoritmos cuando este parámetro se incrementa. Todas las restricciones son binarias, normalizadas y fueron generadas de la siguiente forma $a + X_i \, op \, b + X_j$ con $op = \{<, \leq, >, \geq, =, \neq\}$;

Tabla 6.4: Resultados de las técnicas de arco-consistencia AC3, AC3-OP, AC2001/3.1 y AC2001-OP en problemas binarios, normalizados y no-equilibrados: $\langle 50, 50, m \rangle$

m	**podas**		**AC3**	**AC3-OP**	**AC2001/3.1**	**AC2001-OP**
100	1.06×10^2	*tiempo*	0.01	0.01	0.01	0.01
		cheq	2.10×10^5	1.45×10^5	1.40×10^5	1.40×10^5
		prop	1.97×10^2	4.00×10^0	1.97×10^2	4.00×10^0
200	2.15×10^2	*tiempo*	15.62	15.62	15.62	15.64
		cheq	4.69×10^5	2.91×10^5	2.69×10^5	2.69×10^5
		prop	6.35×10^2	2.00×10^1	6.35×10^2	2.00×10^1
300	3.28×10^2	*tiempo*	15.76	0.01	0.62	0.01
		cheq	7.18×10^5	4.43×10^5	3.87×10^5	3.86×10^5
		prop	1.27×10^3	5.60×10^1	1.27×10^3	5.60×10^1
400	4.45×10^2	*tiempo*	15.64	15.76	15.62	0.01
		cheq	9.63×10^5	6.04×10^5	4.95×10^5	4.93×10^5
		prop	2.09×10^3	1.20×10^2	2.09×10^3	1.20×10^2
500	5.78×10^2	*tiempo*	47	15.68	31.92	15.7
		cheq	1.27×10^6	8.18×10^5	5.91×10^5	5.89×10^5
		prop	3.63×10^3	2.86×10^2	3.63×10^3	2.86×10^2
600	7.20×10^2	*tiempo*	56.16	21.24	52.16	20.32
		cheq	1.60×10^6	1.06×10^6	6.77×10^5	6.74×10^5
		prop	5.61×10^3	5.60×10^2	5.61×10^3	5.60×10^2
700	8.71×10^2	*tiempo*	94	31.5	78.12	31
		cheq	1.96×10^6	1.32×10^6	7.52×10^5	7.48×10^5
		prop	8.01×10^3	9.69×10^2	8.01×10^3	9.69×10^2
800	1.03×10^3	*tiempo*	140.64	46.88	125.02	32.5
		cheq	2.32×10^6	1.57×10^6	8.16×10^5	8.10×10^5
		prop	1.09×10^4	1.54×10^3	1.09×10^4	1.54×10^3

a y $b \in Z$. Se generaron dos tipos de problemas: consistentes e inconsistentes. Se evaluaron 50 casos de pruebas para cada instancia del problema.

Cabe destacar que los algoritmos AC4-OP y AC4 buscan todos los soportes para cada valor del dominio, a diferencia del resto de algoritmos de arco-consistencia que se limitan a buscar un único soporte. La búsqueda de todos los soportes para cada valor hace que el proceso de consistencia sea menos eficiente, pero este tiempo es compensado en el momento de la búsqueda, donde no se requiere repetir el proceso de consistencia, cuando un soporte se torna inválido, como sí ocurre en el resto de algoritmos de consistencia. El tiempo de cómputo (en milisegundos) en los algoritmos AC4 y AC4-OP incluye el tiempo de inicialización y el tiempo de propagación.

Para este tipo de problemas se comparó el desempeño del algoritmo propuesto:

AC4-OP, en relación a su algoritmo base: AC4.

- **Evaluación de problemas arco-consistentes con respecto al número de variables**

La Tabla 6.5 muestra el número de podas (podas), el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq) y el número de propagaciones (prop) en instancias aleatorias, normalizadas y arco-consistentes, donde el número de variables fue incrementado de 50 a 400: $\langle n, 20, 700\rangle$.

Los resultados muestran que el número de chequeos y el número de propagaciones fueron inferiores en AC4-OP en todos los casos con un promedio del 50 % y 20 %, respectivamente, pero AC4 tuvo menor tiempo de cómputo que AC4-OP con un promedio de 24.85 %. Para este tipo de problemas, reducir chequeos y propagaciones aumenta el coste en tiempo de cómputo.

Tabla 6.5: Resultados de las técnicas de arco-consistencia AC4 y AC4-OP en instancias aleatorias, normalizadas y arco-consistentes: $\langle n, 20, 700\rangle$.

		AC4			**AC4-OP**		
n	*podas*	*tiempo*	*cheq*	*prop*	*tiempo*	*cheq*	*prop*
50	872	81	271010	872	87	135618	821
70	768	77	357894	768	75	179035	698
90	747	85	409832	747	139	204992	657
110	730	92	444222	730	85	222174	620
130	724	112	468531	724	243	234321	594
150	713	181	487456	713	366	243775	563
200	711	590	516794	711	780	258436	508
400	704	644	563868	704	701	281954	303

- **Evaluación de problemas arco-consistentes con respecto al número de restricciones**

La Tabla 6.6 muestra el número de podas (podas), el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq) y el número de propagaciones (prop) en instancias aleatorias, normalizadas y arco-consistentes, donde el número de restricciones fue incrementado de 50 a 700: $< 50, 20, m >$.

Los resultados muestran que el número de chequeos de restricciones fue reducido por AC4-OP en todos los casos en un 50 %. Adicionalmente AC4-OP tuvo mejor tiempo de cómputo que AC4 con una mejora de 1.84 %. Para problemas donde se incrementa el número de restricciones, AC4-OP es mejor elección que AC4.

Tabla 6.6: Resultados de las técnicas de consistencia AC4 y AC4-OP en instancias aleatorias, normalizadas y consistentes: $\langle 50, 20, m \rangle$.

		AC4			**AC4-OP**		
m	*podas*	*tiempo*	*cheq*	*prop*	*tiempo*	*cheq*	*prop*
50	53	9	41919	53	7	20980	4
100	106	15	79752	106	14	39906	56
200	215	28	143475	215	25	71785	165
300	329	36	192322	329	36	96224	278
400	446	45	227998	446	44	114076	395
500	580	53	251426	580	54	125806	529
600	720	65	265592	720	64	132901	670
700	871	80	271520	871	81	135872	821

- **Evaluación de problemas inconsistentes con respecto al número de variables**

La Tabla 6.7 muestra el número de podas (podas), el número de chequeos de restricciones (cheq) y el número de propagaciones (prop) en instancias aleatorias, normalizadas e inconsistentes, donde el número de variables fue incrementado de 50 a 150: $\langle n, 20, 800 \rangle$. Los resultados fueron similares a los de los problemas consistentes en relación a que se reduce en un 50 % el número de chequeos de restricciones. El tiempo de cómputo fue inferior en AC4-OP en un 40 % (en promedio). El número de propagaciones fue reducido en un 10 % debido a que el problema se vuelve inconsistente en un número similar de iteraciones. Como puede evidenciarse, AC4-OP es la mejor elección en esta tipología de problemas inconsistentes en todos los casos.

Tabla 6.7: Resultados de las técnicas de arco-consistencia AC4 y AC4-OP en instancias aleatorias, normalizadas e inconsistentes: $\langle n, 20, 800\rangle$.

		AC4			**AC4-OP**		
n	*podas*	*tiempo*	*cheq*	*prop*	*tiempo*	*cheq*	*prop*
50	806	28	112644	806	25	56557	732
70	1133	38	148097	1133	24	74255	1026
90	1431	163	187260	1431	40	93839	1294
110	1753	142	229701	1753	51	115064	1584
130	2053	263	263007	2053	177	131720	1852
150	2341	575	299850	2341	411	150122	2106

- **Evaluación de problemas inconsistentes con respecto a la talla del dominio**

La Tabla 6.8 muestra el número de podas (podas), el número de chequeos de restricciones (cheq) y el número de propagaciones (prop) en instancias aleatorias, normalizadas e inconsistentes, donde la talla de dominio fue incrementada de 20 a 70: $\langle 50, d, 800\rangle$.

Tabla 6.8: Resultados de las técnicas de arco-consistencia AC4 y AC4-OP en instancias aleatorias, normalizadas e inconsistentes: $\langle 50, d, 800\rangle$.

		AC4			**AC4-OP**		
d	*podas*	*tiempo*	*cheq*	*prop*	*tiempo*	*cheq*	*prop*
20	810	19	110674	810	17	55538	735
30	1261	85	338626	1261	56	169816	1187
40	1657	322	753796	1657	141	377652	1583
50	2038	577	1368439	2038	289	685393	1964
60	2382	742	2311487	2382	403	1157344	2308
70	2790	539	3531504	2790	504	1767952	2716

Los resultados fueron similares a los de los problemas consistentes en relación a que se reduce en un 50 % el número de chequeos de restricciones. El tiempo de cómputo fue inferior en AC4-OP en un 38 % (en promedio). El número de propagaciones apenas fue reducido en un 5 % debido a que el problema se vuelve inconsistente en un número similar de iteraciones. Como puede evidenciarse, AC4-OP es la mejor

elección en todos los casos en esta tipología de problemas inconsistentes.

6.1.2. Problemas aleatorios no-normalizados

Las pruebas realizadas en instancias aleatorias, no-normalizadas están caracterizadas por la 5-tupla: $\langle n, d, m, c, p\rangle$, donde n representa el número de variables, d la talla del dominio, m el número de restricciones binarias, c el número máximo de restricciones en cada conjunto de restricciones y p el porcentaje de restricciones no-normalizadas. Las evaluaciones se llevaron a cabo variando un parámetro cada vez. Para cada problema se generaron 50 instancias. Todas las restricciones son binarias y fueron generadas de la siguiente forma $a+X_i\,op\,b+X_j$ con $op = \{<, \leq, >, \geq, =, \neq\}$, $a, b \in Z$. Para este tipo de problemas se generaron instancias consistentes e inconsistentes. Debido al hecho de que estamos trabajando con instancias no-normalizadas, asumimos que cualquier par de variables están restringidas por dos o más restricciones. A este tipo de problemas se le aplicaron técnicas de arco-consistencia y de 2-consistencia para evaluar su rendimiento.

Evaluación de las técnicas de arco-consistencia

- **Evaluación de problemas consistentes con respecto al número de variables**

La Tabla 6.9 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq), el número de propagaciones (prop), el número de podas (podas) y el número de soportes encontrados (soportes) en instancias aleatorias, no-normalizadas y consistentes, al aplicar técnicas de arco-consistencia donde el número de variables fue incrementado de 50 a 150, mientras que la talla del dominio fue fijada a 100, el número de restricciones fue fijado en 700, el número de restricciones no-normalizadas en cada conjunto fue fijado a 4 y el porcentaje de restricciones no-normalizadas fue fijado a 100 %: $\langle n, 100, 700, 4, 100\rangle$. La restringibilidad de los problemas fue del 27 % en promedio.

Los resultados muestran que:

- el número de chequeos de restricciones y el tiempo de cómputo fueron inferiores en AC4-OPNN que en AC4 en todos los casos, en un promedio del 50 % y del

Tabla 6.9: Resultados de las técnicas de arco-consistencia AC6, AC7, AC4, AC4-OPNN y AC3-NN en instancias consistentes, aleatorias y no-normalizadas: $\langle n, 100, 700, 4, 100\rangle$

n		**AC6**	**AC7**	**AC4**	**AC4-OPNN**	**AC3-NN**
50	tiempo	77	11791	5026	4703	16
	cheq	8.39×10^5	1.40×10^5	1.27×10^7	6.38×10^6	8.66×10^5
	prop	44	0	44	44	32
	podas	44	0	44	44	44
	soportes	50	50	9.85×10^6	9.85×10^6	50
70	tiempo	78	11938	5520	5218	16
	cheq	8.38×10^5	1.40×10^5	1.27×10^7	6.37×10^6	8.52×10^5
	prop	33	0	33	33	18
	podas	33	0	33	33	33
	soportes	70	70	9.87×10^6	9.87×10^6	70
90	tiempo	78	12053	6416	5745	16
	cheq	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6	8.42×10^5
	prop	22	0	22	22	8
	podas	22	0	22	22	22
	soportes	90	90	9.88×10^6	9.88×10^6	90
110	tiempo	78	11822	5217	4466	16
	cheq	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6	8.42×10^5
	prop	22	0	22	22	8
	podas	22	0	22	22	22
	soportes	110	110	9.88×10^6	9.88×10^6	110
130	tiempo	81	12106	5878	4970	15
	cheq	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6	8.42×10^5
	prop	22	0	22	22	8
	podas	22	0	22	22	22
	soportes	130	130	9.88×10^6	9.88×10^6	130
150	tiempo	68	12254	6362	5235	6
	cheq	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6	8.42×10^5
	prop	22	0	22	22	8
	podas	22	0	22	22	22
	soportes	150	150	9.88×10^6	9.88×10^6	150

6 %, respectivamente;

- el promedio de valores podados fue de 2.48×10^2 para AC6, AC4, AC4-OPNN y AC3-NN y 0 para AC7. El hecho de que AC7 no realice ninguna poda fue debido a que su estrategia de *recordar* soportes encontrados no funciona en problemas no-normalizados;

- el tiempo de cómputo fue inferior en AC3-NN que en el resto de algoritmos (AC6, AC7, AC4 y AC4-OPNN) en todos los casos. Esto es debido a la baja restringibilidad de las instancias, lo que beneficia al algoritmo de grano-grueso. En el caso de los algoritmos de grano fino, AC6 tuvo mejor tiempo que el resto de algoritmos de grano-fino (AC4 y AC4-OP), debido a la baja restringibilidad de las instancias, ya que cada valor podado realiza propagaciones (es una ventaja de AC6 sobre AC7, AC4 y AC4-OPNN por sus estructuras sencillas); y a que AC4 y AC4-OPNN buscan todos los soportes mientras que AC6 busca un único soporte. AC7 tuvo el peor tiempo de cómputo, lo cual es debido a que el chequeo eficiente (en este caso, la eficiencia se entiende como realizar el menor número de chequeos o evitar chequeos) de AC7 consume tiempo;

- el número de chequeos de restricciones fue menor en AC7 que en el resto de algoritmos (AC6, AC4, AC4-OPNN y AC3-NN) en todos los casos. Sin embargo AC7 no realizó ninguna poda. Esto es debido a que AC7 mantiene estructuras que recuerdan los soportes encontrados. AC7, previo a chequear, revisa estas estructuras y supone que siempre es la misma restricción. Así que en AC7 sólo se verifica que el soporte almacenado pertenezca al dominio de cada variable, lo cual se consigue al revisar la primera de las cuatro restricciones. Lo anterior, unido con la bidireccionalidad al realizar los chequeos, produce que AC7 realice menos chequeos y ninguna poda.

Por ello, AC3-NN es una buena elección sobre el resto de las técnicas de arco-consistencia en problemas consistentes, no-normalizados y poco restringidos, cuando se requiere un único soporte; pero cuando se requieren todos los soportes, AC4-OPNN es la mejor elección.

- **Evaluación de problemas consistentes con respecto al número de restricciones**

La Tabla 6.10 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq), el número de propagaciones (prop), el número de podas (podas) y el número de soportes encontrados (soportes) en instancias aleatorias, no-normalizadas y consistentes, al aplicar técnicas de arco-consistencia donde el número de restricciones fue incrementado de 100 a 800, el número de variables fue fijado a 100, la talla del dominio fue fijada a 100, el número de restricciones no-normalizadas en cada conjunto de restricciones fue fijado en 4 y el porcentaje de restricciones no-normalizadas fue fijado en el 100 %: $\langle 100, 100, m, 4, 100\rangle$. La restringibilidad de los problemas fue del 27 % en promedio.

Los resultados son similares a los problemas cuando variamos el número de variables y muestran que:

- el número de chequeos de restricciones y el tiempo de cómputo fueron inferiores en AC4-OPNN que en AC4 en todos los casos, en un promedio de un 50 % y 11 %, respectivamente;

- el promedio de valores podados fue de 19 para AC6, AC4, AC4-OPNN y AC3-NN y 0 para AC7;

- el tiempo de cómputo fue inferior en AC3-NN que en el resto de algoritmos (AC6, AC7, AC4 y AC4-OPNN) en todos los casos. Esto es debido a la baja restringibilidad de las instancias, lo que beneficia al algoritmo de grano-grueso.

- el número de chequeos de restricciones fue menor en AC7 que en el resto de algoritmos (AC6, AC4, AC4-OPNN y AC3-NN) en todos los casos, pero AC7 no realizó ninguna poda.

Por lo que AC3-NN es de nuevo una buena elección entre las técnicas de arco-consistencia en problemas consistentes, no-normalizados y poco restringidos, cuando se requiere un único soporte. No obstante, cuando se requieren todos los soportes AC4-OPNN es la mejor elección.

Tabla 6.10: Resultados de las técnicas de arco-consistencia AC6, AC7, AC4, AC4-OPNN y AC3-NN en instancias consistentes, aleatorias y no-normalizadas: $\langle 100, 100, m, 4, 100\rangle$

m		**AC6**	**AC7**	**AC4**	**AC4-OP**	**AC3-NN**
100	tiempo	0	274	701	680	0
	cheq	1.02×10^5	2.00×10^4	1.82×10^6	9.12×10^5	1.21×10^5
	prop	11	0	11	11	2
	podas	11	0	11	11	11
	soportes	100	100	4.54×10^5	1.82×10^6	100
200	tiempo	16	1027	1500	1397	0
	cheq	2.40×10^5	4.01×10^4	3.64×10^6	7.82×10^3	2.40×10^5
	prop	11	0	11	11	2
	podas	11	0	11	11	11
	soportes	100	100	9.09×10^5	9.09×10^5	100
300	tiempo	31	2277	2343	2041	0
	cheq	3.59×10^5	6.01×10^4	5.46×10^6	2.73×10^6	3.59×10^5
	prop	22	0	11	11	2
	podas	22	0	11	11	11
	soportes	100	100	1.36×10^6	1.36×10^6	100
400	tiempo	47	4010	3332	2809	2
	cheq	4.79×10^5	8.01×10^4	7.28×10^6	3.64×10^6	4.84×10^5
	prop	22	0	22	22	8
	podas	22	0	22	22	22
	soportes	100	100	1.81×10^6	1.81×10^6	100
500	tiempo	62	6223	3938	3460	16
	cheq	5.99×10^5	1.00×10^5	9.10×10^6	4.55×10^6	6.03×10^5
	prop	22	0	22	22	8
	podas	22	0	22	22	22
	soportes	100	100	2.27×10^6	2.27×10^6	100
600	tiempo	71	8927	4572	4083	16
	cheq	7.18×10^5	1.20×10^5	1.09×10^7	5.46×10^6	7.23×10^5
	prop	22	0	22	22	8
	podas	22	0	22	22	22
	soportes	100	100	2.72×10^6	2.72×10^6	100
700	tiempo	78	12038	5393	4902	16
	cheq	8.37×10^5	1.40×10^5	1.27×10^7	6.37×10^6	8.42×10^5
	prop	22	0	22	22	8
	podas	22	0	22	22	22
	soportes	100	100	3.18×10^6	3.18×10^6	100
800	tiempo	93	15623	6224	5649	16
	cheq	9.57×10^5	1.60×10^5	1.46×10^7	7.28×10^6	9.71×10^5
	prop	33	0	33	33	18
	podas	33	0	33	33	33
	soportes	100	100	3.62×10^6	3.62×10^6	100

- **Evaluación de problemas inconsistentes con respecto al número de variables**

La Tabla 6.11 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos (cheq), el número de propagaciones (prop) y el número de podas (podas) en instancias inconsistentes al aplicar técnicas de arco-consistencia, donde el número de variables fue incrementado de 90 a 190; la talla del dominio fue fijada a 100; el número de restricciones fue fijado a 700; el número de restricciones no-normalizas por cada conjunto fue fijado en 4; y el porcentaje de restricciones no-normalizadas fue fijado a 100 %: $\langle n, 100, 700, 4, 100\rangle$. La restringibilidad de los problemas fue del 60 % en promedio.

Tabla 6.11: Resultados de las técnicas de arco-consistencia AC6, AC7, AC4, AC4-OPNN y AC3-NN en instancias inconsistentes, aleatorias y no-normalizadas: $\langle n, 100, 700, 4, 100\rangle$

n		**AC6**	**AC7**	**AC4**	**AC4-OPNN**	**AC3-NN**
90	tiempo	8364	1218	2695	2360	445
	cheq	3.75×10^6	5.61×10^5	1.10×10^7	5.52×10^6	9.87×10^6
	prop	6.84×10^3	6.45×10^4	7.26×10^3	7.12×10^3	1.17×10^4
	podas	6.84×10^3	3.92×10^3	7.26×10^3	7.26×10^3	8.83×10^3
110	tiempo	10651	1798	2377	2029	479
	cheq	3.91×10^6	7.14×10^5	1.16×10^7	5.81×10^6	1.14×10^7
	prop	7.51×10^3	9.17×10^4	8.61×10^3	8.44×10^3	1.28×10^4
	podas	7.51×10^3	4.97×10^3	8.61×10^3	8.61×10^3	1.07×10^4
130	tiempo	17832	2936	2649	2228	555
	cheq	4.20×10^6	8.40×10^5	1.21×10^7	6.05×10^6	1.35×10^7
	prop	8.39×10^3	1.38×10^5	9.22×10^3	9.02×10^3	1.47×10^4
	podas	8.39×10^3	5.96×10^3	9.22×10^3	9.22×10^3	1.26×10^4
150	tiempo	14490	3748	2720	2289	581
	cheq	4.20×10^6	9.80×10^5	1.24×10^7	6.21×10^6	1.51×10^7
	prop	9.31×10^3	1.43×10^5	1.07×10^4	1.04×10^4	1.57×10^4
	podas	9.31×10^3	8.23×10^3	1.07×10^4	1.07×10^4	1.45×10^4
170	tiempo	17300	7565	2878	2405	653
	cheq	4.38×10^6	1.17×10^6	1.27×10^7	6.35×10^6	1.70×10^7
	prop	1.04×10^4	2.85×10^5	1.09×10^4	1.07×10^4	1.80×10^4
	podas	1.04×10^4	1.06×10^4	1.09×10^4	1.09×10^4	1.62×10^4
190	tiempo	19139	9173	3020	2453	624
	cheq	4.41×10^6	1.30×10^6	1.28×10^7	6.43×10^6	1.73×10^7
	prop	1.15×10^4	3.07×10^5	1.26×10^4	1.23×10^4	1.71×10^4
	podas	1.15×10^4	9.62×10^3	1.26×10^4	1.26×10^4	1.79×10^4

Los resultados muestran que AC4-OPNN fue más rápido que el resto de los algoritmos (AC6, AC7 y AC4) en promedio, porque AC4-OPNN realiza bidireccionalmente la búsqueda de todos los soportes, lo cual hace que AC4-OPNN sea una buena elección para este tipo de problemas. Adicionalmente, AC4-OPNN y AC4 realizan más podas que AC6 y AC7. Nuevamente, AC7 realiza menos chequeos de restricciones que AC6, AC4 y AC4-OPNN, pero invirtiendo más tiempo que AC4 y AC4-OPNN. AC6 tuvo peores resultados en este tipo de problemas, i.e., con $n = 190$, AC6 necesitó 20 segundos para llevar a acabo la arco-consistencia, mientras que AC7 requirió casi 10 segundos (la bidirectionalidad de AC7 mejora los resultados con relación a AC6) y, AC4-OPNN requirió 1,9 y AC4 requirió 2,5 segundos (la bidireccionalidad de AC4-OPNN fue la mejor elección de las técnicas de arco-consistencia). También, AC4-OPNN realizó menos chequeos y menos propagaciones que AC4.

- **Evaluación de problemas inconsistentes con respecto al número de restricciones**

La Tabla 6.12 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos (cheq), el número de propagaciones (prop) y el número de podas (podas) en instancias inconsistentes al aplicar técnicas de arco-consistencia, donde el número de restricciones fue incrementado de 100 a 800; el número de variables fue fijado en 100; la talla del dominio fue fijada en 100; el número de restricciones no-normalizas por cada conjunto fue fijado en 4; y el porcentaje de restricciones no-normalizadas fue fijado en 100 %: $\langle 100, 100, m, 4, 100\rangle$. La restringibilidad de las instancias fue del 60 %. Como en la tabla anterior, los resultados fueron similares. Los resultados muestran que AC4-OPNN tuvo menor tiempo de cómputo promedio que el resto de los algoritmos (AC4-OPNN fue más rápido que AC4 (21 %), AC6 (80 %) y AC7 (24 %)). También, AC4-OPNN realizó menos chequeos de restricciones y menos propagaciones que AC4.

- **Evaluación de problemas inconsistentes con respecto a la talla del dominio**

La Tabla 6.13 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq), el número de propagaciones (prop) y el número

Tabla 6.12: Resultados de las técnicas de arco-consistencia AC6, AC7, AC4, AC4-OPNN y AC3-NN en instancias inconsistentes, aleatorias y no-normalizadas: $\langle 100, 100, m, 4, 100\rangle$

m		**AC6**	**AC7**	**AC4**	**AC4-OPNN**	**AC3-NN**
100	tiempo	1744	1491	370	339	88
	cheq	8.60×10^5	6.17×10^5	2.00×10^6	1.00×10^6	5.50×10^6
	prop	5.60×10^3	9.75×10^4	5.36×10^3	5.22×10^3	4.63×10^3
	podas	5.60×10^3	4.65×10^3	5.36×10^3	5.36×10^3	5.90×10^3
200	tiempo	6299	1330	730	649	157
	cheq	1.59×10^6	6.34×10^5	3.90×10^6	1.95×10^6	8.73×10^6
	prop	6.65×10^3	7.55×10^4	6.02×10^3	5.87×10^3	8.18×10^3
	podas	6.65×10^3	5.22×10^3	6.02×10^3	6.02×10^3	8.23×10^3
300	tiempo	1001	703	1033	938	198
	cheq	1.83×10^6	5.88×10^5	5.65×10^6	2.83×10^6	9.77×10^6
	prop	6.29×10^3	7.52×10^4	7.13×10^3	6.97×10^3	9.49×10^3
	podas	6.29×10^3	2.30×10^3	7.13×10^3	7.13×10^3	9.28×10^3
400	tiempo	5705	423	1432	1215	248
	cheq	2.52×10^6	5.75×10^5	7.31×10^6	3.66×10^6	1.02×10^7
	prop	5.92×10^3	9.86×10^3	6.82×10^3	6.67×10^3	9.71×10^3
	podas	5.92×10^3	3.11×10^3	6.82×10^3	6.82×10^3	9.52×10^3
500	tiempo	5723	980	1833	1533	333
	cheq	2.99×10^6	6.17×10^5	8.79×10^6	4.40×10^6	1.11×10^7
	prop	6.20×10^3	4.98×10^4	7.06×10^3	6.91×10^3	1.16×10^4
	podas	6.20×10^3	3.98×10^3	7.06×10^3	7.06×10^3	9.71×10^3
600	tiempo	6731	1901	2160	1815	389
	cheq	3.38×10^6	6.51×10^5	1.01×10^7	5.07×10^6	1.06×10^7
	prop	6.33×10^3	1.08×10^5	7.44×10^3	7.29×10^3	1.18×10^4
	podas	6.33×10^3	4.49×10^3	7.44×10^3	7.44×10^3	9.72×10^3
700	tiempo	5868	1853	2453	2090	462
	cheq	3.76×10^6	6.45×10^5	1.14×10^7	5.68×10^6	1.05×10^7
	prop	6.81×10^3	9.72×10^4	7.66×10^3	7.51×10^3	1.21×10^4
	podas	6.81×10^3	4.67×10^3	7.66×10^3	7.66×10^3	9.75×10^3
800	tiempo	7013	2104	2747	2364	526
	cheq	4.12×10^6	6.58×10^5	1.24×10^7	6.22×10^6	1.02×10^7
	prop	7.08×10^3	1.17×10^5	8.05×10^3	7.90×10^3	1.21×10^4
	podas	7.08×10^3	4.63×10^3	8.05×10^3	8.05×10^3	9.77×10^3

de podas (podas) en instancias aleatorias, no-normalizadas e inconsistentes al aplicar técnicas de arco-consistencia, donde la talla del dominio fue incrementada de 50 a 300, el número de variables fue fijado en 200, el número de restricciones fue fijado en 500, el máximo número de restricciones no-normalizadas en cada conjunto fue fijado en 4 y el porcentaje de restricciones no-normalizadas fue fijada en 100 %: $\langle 200, d, 500, 4, 100\rangle$. En todas las instancias los valores del dominio fueron generados al azar y la restringibilidad promedio fue fijada al 60 %.

Tabla 6.13: Resultados de las técnicas de arco-consistencia AC6, AC7, AC4, AC4-OPNN y AC3-NN en instancias inconsistentes, aleatorias y no-normalizadas: $\langle 200, d, 500, 4, 100\rangle$

d		**AC6**	**AC7**	**AC4**	**AC4-OPNN**	**AC3-NN**
50	tiempo	2010	2165	521	445	96
	cheq	8.17×10^5	3.33×10^5	2.28×10^6	1.14×10^6	1.15×10^6
	prop	5.69×10^3	5.84×10^4	5.98×10^6	5.66×10^3	6.64×10^3
	podas	5.69×10^3	5.34×10^3	5.98×10^3	5.98×10^3	8.58×10^3
100	tiempo	3380	8628	2003	1541	390
	cheq	3.11×10^6	1.35×10^6	9.55×10^6	4.78×10^6	9.20×10^6
	prop	1.10×10^4	2.83×10^5	1.20×10^4	1.17×10^4	1.87×10^4
	podas	1.10×10^4	1.15×10^4	1.20×10^4	1.20×10^4	1.73×10^4
150	tiempo	48680	14780	4684	3586	681
	cheq	8.05×10^6	2.91×10^6	2.18×10^7	1.09×10^7	2.38×10^7
	prop	1.71×10^4	5.02×10^5	1.73×10^4	1.69×10^4	2.38×10^4
	podas	1.71×10^4	1.51×10^4	1.73×10^4	1.73×10^4	2.64×10^4
200	tiempo	137568	38380	7901	6691	1280
	cheq	1.56×10^7	5.47×10^6	3.91×10^7	1.95×10^7	5.58×10^7
	prop	2.16×10^4	1.33×10^6	1.99×10^4	1.95×10^4	3.40×10^4
	podas	2.16×10^4	2.21×10^4	1.99×10^4	1.99×10^4	3.54×10^4
250	tiempo	67092	37935	13681	12905	2158
	cheq	2.06×10^7	8.03×10^6	6.14×10^7	3.07×10^7	1.05×10^8
	prop	2.99×10^4	1.28×10^6	2.85×10^4	2.82×10^4	4.30×10^4
	podas	2.99×10^4	2.47×10^4	2.85×10^4	2.85×10^4	4.41×10^4
300	tiempo	370638	31901	18583	17271	4439
	cheq	3.78×10^7	1.05×10^7	8.87×10^7	4.43×10^7	2.35×10^8
	prop	3.51×10^4	1.06×10^6	3.75×10^4	3.72×10^4	6.61×10^4
	podas	3.51×10^4	1.91×10^4	3.75×10^4	3.75×10^4	5.33×10^4

Los resultados muestran que AC3-NN fue el que tuvo el menor tiempo de cómputo, inferior a un orden de magnitud (con relación a AC7, AC4 y AC4-OP) e inferior a dos órdenes de magnitud (en relación a AC6). También muestran que AC4-OPNN mantuvo un tiempo de cómputo menor que el resto de algoritmos de grano-fino

(AC4-OPNN fue más rápido que AC4 (14 %), AC6 (95 %) y AC7 (75 %)). Las mejoras de AC4 y AC4-OPNN en relación a AC6 y AC7 son debidas al hecho que AC6 y AC7 deben realizar un ordenamiento del dominio antes de realizar el proceso de consistencia. Un orden ascendente de los dominios es obligatorio para AC6 y AC7 pero no lo es para AC4 y AC4-OPNN. El ordenamiento de los dominios fue llevado a cabo utilizando el algoritmo quicksort. El algoritmo quicksort añade una complejidad de $O(n.d.\lg(d))$ a AC6 y AC7, donde n es el número de variables y d la talla del dominio en el problema.

Evaluación de las técnicas de 2-consistencia

En la evaluación de las técnicas de 2-consistencia utilizamos las mismas instancias no-normalizadas que se generaron para evaluar las técnicas de arco-consistencia, lo cual nos permite hacer comparaciones entre las técnicas de arco-consistencia (como referencia) y las de 2-consistencia.

- **Evaluación de problemas consistentes con respecto al número de variables**

La Tabla 6.14 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq), el número de propagaciones (prop), el número de podas (podas) y el número de soportes encontrados (soportes) en instancias aleatorias, no-normalizadas y consistentes, al aplicar técnicas de 2-consistencia donde el número de variables fue incrementado de 50 a 190 y la talla del dominio fue fijada a 100, el número de restricciones fue fijado en 700, el número de restricciones no-normalizadas en cada conjunto fue fijado en 4 y el porcentaje de restricciones no-normalizadas fue fijado en 100 %: $\langle n, 100, 700, 4, 100\rangle$. La restringibilidad de los problemas fue del 27 % en promedio.

Los resultados muestran que:

- el número de chequeos de restricciones y el tiempo de cómputo fueron inferiores en todas las técnicas de 2-consistencia (2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6) en relación a realizar el proceso de normalización y arco-consistencia (AC3NH);
- todos los algoritmos de 2-consistencia alcanzan la misma cantidad de podas. Así, el promedio general de valores podados fue de 231 para las técnicas de

Tabla 6.14: Resultados de las técnicas de 2-consistencia: AC3NH, 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6 en instancias aleatorias, consistentes y no-normalizadas: $\langle n, 100, 700, 4, 100\rangle$

n		**AC3NH**	**2-C3**	**2-C3OP**	**2-C3OPL**	**2-C4**	**2-C6**
50	tiempo	89438 (122)	109	109	99	721	117
	setChecks	-	356	181	181	175	350
	cheq	7.14×10^6	1.30×10^6	9.96×10^5	9.65×10^5	2.90×10^6	1.00×10^6
	prop	3.27×10^7	6	6	6	150	250
	podas	250	250	250	250	250	250
	soportes	50	50	50	50	8.77×10^5	50
70	tiempo	89409 (127)	109	109	109	729	125
	setChecks	-	353	178	178	175	350
	cheq	7.14×10^6	1.27×10^6	9.73×10^5	9.58×10^5	2.90×10^6	9.93×10^5
	prop	3.20×10^7	3	3	3	100	220
	podas	220	220	220	220	220	220
	soportes	70	70	70	70	8.85×10^5	70
90	tiempo	89446 (126)	109	109	108	751	121
	setChecks	-	351	176	176	175	350
	cheq	7.14×10^6	1.25×10^6	9.55×10^5	9.50×10^5	2.89×10^6	9.85×10^5
	prop	3.16×10^7	1	1	1	50	190
	podas	190	190	190	190	190	190
	soportes	90	90	90	90	8.90×10^5	90
110	tiempo	89425 (122)	109	108	96	648	113
	setChecks	-	351	176	176	175	350
	cheq	7.14×10^6	1.25×10^6	9.56×10^5	9.51×10^5	2.89×10^6	9.86×10^5
	prop	3.16×10^7	1	1	1	50	210
	podas	210	210	210	210	210	210
	soportes	110	110	110	110	8.90×10^5	110
130	tiempo	89451 (119)	109	108	109	679	124
	setChecks	-	351	176	176	175	350
	cheq	7.14×10^6	1.25×10^6	9.57×10^5	9.52×10^5	2.89×10^6	9.87×10^5
	prop	3.16×10^7	1	1	1	50	230
	podas	230	230	230	230	230	230
	soportes	130	130	130	130	8.90×10^5	130
150	tiempo	89482 (119)	97	98	111	662	109
	setChecks	-	351	176	176	175	350
	cheq	7.14×10^6	1.25×10^6	9.58×10^5	9.53×10^5	2.89×10^6	9.88×10^5
	prop	3.16×10^7	1	1	1	50	250
	podas	250	250	250	250	250	250
	soportes	150	150	150	150	8.90×10^5	150
170	tiempo	89470 (118)	95	94	124	674	110
	setChecks	-	351	176	176	175	350
	cheq	7.14×10^6	1.25×10^6	9.59×10^5	9.54×10^5	2.89×10^6	9.89×10^5
	prop	3.16×10^7	1	1	1	50	270
	podas	270	270	270	270	270	270
	soportes	170	170	170	170	8.90×10^5	170
190	tiempo	89395 (120)	96	94	110	725	111
	setChecks	-	350	175	175	175	350
	cheq	7.14×10^6	1.23×10^6	9.45×10^5	9.45×10^5	2.88×10^6	9.80×10^5
	prop	3.14×10^7	0	0	0	0	226
	podas	226	226	226	226	226	226
	soportes	190	190	190	190	8.93×10^5	190

2-consistencia (un orden de magnitud por encima de las técnicas de arco-consistencia), mientras que las técnicas de arco-consistencia (ver Tabla 6.9) tienen en promedio general una poda de 24.8 (para aquellas que lograron realizar poda). Esto es debido a que en los problemas se mantiene la talla de dominio y las restricciones tienen operadores $(=, \neq, \leq, \geq)$, de forma que los algoritmos de arco-consistencia no podan ningún valor analizando las restricciones individualmente. Sin embargo, las técnicas de 2-consistencia analizan los conjuntos de restricciones que mezclan estos operadores y son capaces de podar más espacio de búsqueda.

- el tiempo de cómputo (milisegundos) fue bastante similar en las técnicas de 2-consistencia de grano-grueso: 2-C3 (104.33), 2-C3OP (103.75) y 2-C3OPL (108.29) lo cual se debe a la baja restringibilidad de las instancias. No obstante, las técnicas de 2-consistencia de grano fino variaron en cuanto al tiempo: 2-C6 (116.15) y 2-C4 (698.72), lo cual se debe a que 2-C6 se limita a la búsqueda de un sólo soporte (120 en promedio) y 2-C4 busca todos los soportes (8.88×10^5 en promedio). En relación con las técnicas de arco-consistencia, el tiempo de cómputo de las técnicas de 2-consistencia fue superior (80 % en promedio de 2-C3OP en relación con AC3-NN, donde ambas técnicas son las más rápidas en 2-consistencia y arco-consistencia, respectivamente), pero también la 2-consistencia fue superior en la cantidad de podas realizadas (2-C3OP realizó 480 % mas de poda que AC3-NN). Si se compara 2-C3 en relación a AC4 y AC7 (ver Tabla 6.9), se puede observar que 2-C3 fue más rápido en promedio en un 98 % que AC4 (6073 milisegundos en promedio) y en 99 % que AC7 (12046 milisegundos en promedio).

- el número de conjuntos de restricciones chequeados fue inferior en 2-C4 (175 en promedio) que en el resto de técnicas de 2-consistencia, lo cual se debe a su estrategia de realizar el chequeo de los conjuntos de restricciones una única vez. El promedio de chequeos de conjuntos de restricciones en las técnicas 2-C3OP y 2-C3OPL fue el mismo 177 (ya que hubo muy poca propagación), mientras que 2-C3 y 2-C6 tuvieron en promedio 352 y 350 respectivamente. Esto se debe a que 2-C4, 2-C3OP y 2-C3OPL realizan el proceso de 2-consistencia bidireccionalmente y al haber poca propagación, estos algoritmos -sin importar

su granularidad- realizan un proceso similar de conjuntos de restricciones, mientras que 2-C3 y 2-C6 realizaron un 50 % más de revisiones de conjuntos de restricciones.

Comparando los resultados con los mostrados en la Tabla 6.9 observamos que las técnicas de 2-consistencia son una mejor elección que las de arco-consistencia en instancias no-normalizadas. Así, 2-C3OP es una buena elección sobre el resto de las técnicas de 2-consistencia en problemas consistentes, no-normalizados y poco restringidos, cuando se requiere un único soporte; pero cuando se requieren todos los soportes, 2-C4 es la mejor elección.

- **Evaluación de problemas consistentes con respecto al número de restricciones**

La Tabla 6.15 muestra el tiempo de cómputo en milisegundos (tiempo), el número de conjunto de restricciones (setChecks), el número de chequeos de restricciones (cheq), el número de propagaciones (prop), el número de podas (podas) y el número de soportes encontrados (soportes) en instancias aleatorias, no-normalizadas y consistentes, al aplicar técnicas de 2-consistencia donde el número de restricciones fue incrementado de 100 a 800, el número de variables fue fijado a 100, la talla del dominio fue fijada a 100, el número de restricciones no-normalizadas en cada conjunto fue fijado en 4 y el porcentaje de restricciones no-normalizadas fue del 100 %: $\langle 100, 100, m, 4, 100\rangle$. La restringibilidad de los problemas fue del 27 % en promedio.

Los resultados muestran que:

- el promedio de valores podados por todos los algoritmos de 2-consistencia fue de 169, mientras que los algoritmos de arco-consistencia que realizaron podas su promedio de valores podados fue de 19 (ver Tabla 6.10);

- el tiempo de cómputo promedio (en milisegundos) fue inferior en 2-C3OPL (66.19) que en el resto de algoritmos: 2-C3 (70.31) , 2-C3OP (70.3), 2-C4 (424.62), 2-C6 (73.89) y AC3NH (57234). Esto es debido a la baja restringibilidad de las instancias y al aumento del número de restricciones, lo que beneficia a la estrategia de 2-C3OPL por realizar bidireccionalmente los chequeos y evaluar la mitad de los conjuntos de restricciones. En relación a los

Tabla 6.15: Resultados de las técnicas de 2-consistencia: AC3NH, 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6 en instancias aleatorias, consistentes y no-normalizadas: $\langle 100, 100, m, 4, 100\rangle$

m		**AC3NH**	**2-C3**	**2-C3OP**	**2-C3OPL**	**2-C4**	**2-C6**
100	tiempo	12476 (20)	16	16	16	117	16
	setChecks	-	50	25	25	25	50
	cheq	1.02×10^6	1.89×10^5	1.43×10^5	1.43×10^5	4.20×10^5	1.48×10^5
	prop	4.48×10^6	0	0	0	0	76
	podas	76	76	76	76	76	76
	soportes	100	100	100	100	1.27×10^5	100
200	tiempo	25079 (37)	31	31	31	203	31
	setChecks	-	100	50	50	50	100
	cheq	2.04×10^6	3.63×10^5	2.77×10^5	2.77×10^5	8.30×10^5	2.87×10^5
	prop	8.96×10^6	0	0	0	0	101
	podas	101	101	101	101	101	101
	soportes	100	100	100	100	2.55×10^5	100
300	tiempo	37744 (57)	47	47	47	301	47
	setChecks	-	150	75	75	75	150
	cheq	3.06×10^6	5.37×10^5	4.10×10^5	4.10×10^5	1.24×10^6	4.25×10^5
	prop	1.34×10^7	0	0	0	0	126
	podas	126	126	126	126	126	126
	soportes	100	100	100	100	4.20×10^5	100
400	tiempo	50593 (75)	62	62	62	376	63
	setChecks	-	201	101	101	100	200
	cheq	4.08×10^6	7.32×10^5	5.58×10^5	5.53×10^5	1.66×10^6	5.73×10^5
	prop	1.81×10^7	1	1	1	50	200
	podas	200	200	200	200	200	200
	soportes	100	100	100	100	4.20×10^5	100
500	tiempo	63445 (92)	78	78	78	463	78
	setChecks	-	251	126	126	125	250
	cheq	5.10×10^6	9.04×10^5	6.90×10^5	6.85×10^5	2.07×10^6	7.10×10^5
	prop	2.26×10^7	1	1	1	50	200
	podas	200	200	200	200	200	200
	soportes	100	100	100	100	4.20×10^5	100
600	tiempo	76426 (108)	94	94	93	557	98
	setChecks	-	301	151	151	150	300
	cheq	6.12×10^6	1.08×10^6	8.23×10^5	8.18×10^5	2.48×10^6	8.48×10^5
	prop	2.71×10^7	1	1	1	50	200
	podas	200	200	200	200	200	200
	soportes	100	100	100	100	4.20×10^5	100
700	tiempo	89500 (127)	109	109	94	642	123
	setChecks	-	351	176	176	175	350
	cheq	7.14×10^6	1.25×10^6	9.55×10^5	9.50×10^5	2.89×10^6	9.85×10^5
	prop	3.16×10^7	1	1	1	50	200
	podas	200	200	200	200	200	200
	soportes	100	100	100	100	4.20×10^5	100
800	tiempo	102611 (147)	125	125	109	739	136
	setChecks	-	403	203	203	200	400
	cheq	8.16×10^6	1.45×10^6	1.11×10^6	1.09×10^6	3.31×10^6	1.13×10^6
	prop	3.65×10^7	3	3	3	100	250
	podas	250	250	250	250	250	250
	soportes	100	100	100	100	4.20×10^5	100

algoritmos de arco-consistencia, estos pueden llegar a ser más rápidos (ejemplo, un 87 % AC3-NN en relación a 2-C3OPL) pero con el coste de que realizan menos podas (89 % menos).

- El número de chequeos de restricciones promedio fue menor en 2-C3OPL (6.16×10^5) que en el resto de algoritmos de 2-consistencia: 2-C3 (8.12×10^5), 2-C3OP (6.20×10^5), 2-C4 (1.86×10^6), 2-C6 (6.38×10^5) y AC3NH (4.59×10^6).
- El número de conjunto de restricciones procesados fue menor en 2-C4 que en el resto de algoritmos de 2-consistencia, ya que 2-C4 procesa una única vez cada conjunto de restricciones. En orden descendente le siguen con valores muy cercanos 2-C3OP y 2-C3OPL, ya que ambos procesan solamente conjuntos de restricciones directas (por realizar el proceso bidireccionalmente) y al haber poca propagación sus promedios se acercan al de 2-C4. 2-C3 y 2-C6 procesaron el doble de conjunto de restricciones que el resto de algoritmos.
- De forma similar a los resultados de la Tabla 6.14, el número de conjuntos evaluados fue inferior en los algoritmos 2-C3OPL, 2-C3OP, 2-C4 (113 en promedio) que en 2-C3 (226 en promedio); 2-C6 realizó en promedio 225 evaluaciones. Nuevamente, se benefician los algoritmos de grano-grueso por la poca propagación realizada.
- El proceso de normalización con arco-consistencia (AC3NH) pese a alcanzar la misma cantidad de podas, fue el que tuvo peor desempeño.

De nuevo, la 2-consistencia es mejor opción en la arco-consistencia en este tipo de instancias y 2-C3OPL es una buena elección entre las técnicas de 2-consistencia en problemas consistentes, no-normalizados y poco restringidos, cuando se requiere un único soporte. De nuevo, cuando se requieren todos los soportes, 2-C4 es la mejor elección.

- **Evaluación de problemas inconsistentes con respecto al número de variables**

La Tabla 6.16 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos (cheq), el número de propagaciones (prop) y el número de podas (podas) en instancias inconsistentes al aplicar técnicas de 2-consistencia, donde el número

de variables fue incrementado de 90 a 190; la talla del dominio fue fijada a 100; el número de restricciones fue fijado a 700; el número de restricciones no-normalizas por cada conjunto fue fijado en 4; y el porcentaje de restricciones no-normalizadas fue del 100 %: $\langle n, 100, 700, 4, 100\rangle$. La restringibilidad de los problemas fue del 60 % en promedio.

Tabla 6.16: Resultados de las técnicas de 2-consistencia: 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6 en instancias aleatorias, inconsistentes y no-normalizadas: $\langle n, 100, 700, 4, 100\rangle$

n		**2-C3**	**2-C3OP**	**2-C3OPL**	**2-C4**	**2-C6**
90	tiempo	1677	706	540	2795	625
	cheq	1.97×10^7	4.47×10^6	3.30×10^6	5.52×10^6	3.75×10^6
	prop	1.17×10^4	2.59×10^3	2.59×10^3	7.12×10^3	6.84×10^3
	podas	8.83×10^3	8.27×10^3	8.27×10^3	7.26×10^3	6.85×10^3
110	tiempo	1705	733	559	2545	683
	cheq	2.28×10^7	5.26×10^6	3.62×10^6	5.81×10^6	3.91×10^6
	prop	1.28×10^4	2.99×10^3	2.99×10^3	8.44×10^3	7.51×10^3
	podas	1.07×10^4	1.01×10^4	1.01×10^4	8.61×10^3	7.51×10^3
130	tiempo	2076	882	633	2717	854
	cheq	2.70×10^7	6.23×10^6	4.00×10^6	6.05×10^6	4.20×10^6
	prop	1.47×10^4	3.48×10^3	3.48×10^3	9.02×10^3	8.39×10^3
	soportes	1.26×10^4	1.20×10^4	1.20×10^4	9.22×10^3	8.39×10^3
150	tiempo	2290	1062	701	2746	788
	cheq	3.01×10^7	7.37×10^6	4.32×10^6	6.21×10^6	4.20×10^6
	prop	1.57×10^4	4.08×10^3	4.07×10^3	1.04×10^4	9.31×10^3
	podas	1.45×10^4	1.39×10^4	1.39×10^4	1.07×10^4	9.31×10^3
170	tiempo	2620	1151	740	2823	858
	cheq	3.41×10^7	8.04×10^6	4.57×10^6	6.35×10^6	4.38×10^6
	prop	1.80×10^4	4.29×10^3	4.28×10^3	1.07×10^4	1.04×10^4
	podas	1.62×10^4	1.55×10^4	1.55×10^4	1.09×10^4	1.04×10^4
190	tiempo	2638	1328	814	2859	897
	cheq	3.45×10^7	9.22×10^6	4.92×10^6	6.43×10^6	4.41×10^6
	prop	1.71×10^4	4.89×10^3	4.89×10^3	1.23×10^4	1.15×10^4
	podas	1.79×10^4	1.71×10^4	1.71×10^4	1.26×10^4	1.15×10^4

Los resultados muestran que 2-C3OPL fue más rápido que el resto de los algoritmos de 2-consistencia: 2-C3, 2-C3OP, 2-C4 y 2-C6 en promedio, porque 2-C3OPL evita chequeos innecesarios y realiza bidireccionalmente la búsqueda de todos los soportes, lo cual hace que 2-C3OPL sea una buena elección para este tipo de problemas. Al tratarse de problemas sin solución, la menor cantidad de podas indica que el algoritmo debe realizar menos chequeos y menos propagaciones. En este sentido,

2-C6 fue el algoritmo que realizó menos podas (7.93×10^3 en promedio), seguido por 2-C4 (8.71×10^3 en promedio). 2-C3OPL y 2-C3OP realizaron la misma cantidad de podas (1.10×10^4 en promedio) y 2-C3 realizó el mayor número de podas (1.16×10^4 en promedio).

El número de propagaciones promedio de los algoritmos de 2-consistencia (en orden creciente) fue el siguiente: 2-C3OPL y 2-C3OP (3.28×10^3), 2-C6 (7.93×10^3), 2-C4 (8.53×10^3) y 2-C3 (1.32×10^4). Comparando con las propagaciones realizadas por los algoritmos de arco-consistencia, 2-C3OPL y 2-C6 realizaron menos cantidad de propagaciones que estos. Nuevamente, los resultados indican que 2-C3OPL realiza menos chequeos de restricciones que el resto de algoritmos de 2-consistencia, lleva a cabo menos propagaciones y evalúa menos conjuntos de restricciones, por lo que invierte menos tiempo en detectar que el problema es inconsistente y finalizar el proceso. Los tiempos promedios (en milisegundos) de los algoritmos de 2-consistencia en esta tipología de problemas en orden creciente fueron: 2-C3OPL (607.56), 2-C6 (699.58), 2-C3OP (862.2), 2-C3 (1899.95), 2-C4 (2617.84). Comparando con los resultados obtenidos en los algoritmos de arco-consistencia, los algoritmos de 2-consistencia 2-C3OPL, 2-C6, 2-C3OP y 2-C3 fueron más rápidos que AC4, AC6 y AC7. 2-C4 fue más rápido que AC6 y AC7. No obstante AC3-NN fue un 19 % más rápido que 2-C3OPL, pero AC3-NN realizó una mayor cantidad de chequeos que 2-C3OPL (el número de chequeos realizados por AC3-NN fue superior en un orden de magnitud a los realizados por 2-C3OPL). Debido a que AC3-NN no garantiza que detecte la inconsistencia en problemas no-normalizados, 2-C3OPL es la mejor elección en esta tipología de problemas.

- **Evaluación de problemas inconsistentes con respecto al número de restricciones**

La Tabla 6.17 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos (cheq), el número de propagaciones (prop) y el número de podas (podas) en instancias inconsistentes al aplicar técnicas de 2-consistencia, donde el número de restricciones fue incrementado de 100 a 800; el número de variables y la talla del dominio fueron fijadas en 100; el número de restricciones no-normalizas por cada conjunto fue fijado en 4; y el porcentaje de restricciones no-normalizadas fue fijado al 100 %: $\langle 100, 100, m, 4, 100 \rangle$. La restringibilidad de las instancias fue del 60 %.

Tabla 6.17: Resultados de las técnicas de 2-consistencia: 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6 en instancias aleatorias, inconsistentes y no-normalizadas: $\langle 100, 100, m, 4, 100\rangle$

m		**2-C3**	**2-C3OP**	**2-C3OPL**	**2-C4**	**2-C6**
100	tiempo	639	1013	404	426	250
	cheq	1.10×10^7	9.25×10^6	3.43×10^6	1.00×10^6	8.60×10^5
	prop	4.63×10^3	5.19×10^3	5.16×10^3	5.22×10^3	5.60×10^3
	podas	5.90×10^3	5.94×10^3	5.93×10^3	5.36×10^3	5.60×10^3
200	tiempo	1009	600	303	810	349
	cheq	1.75×10^7	5.40×10^6	2.49×10^6	1.95×10^6	1.59×10^6
	prop	8.18×10^3	2.88×10^3	2.86×10^3	5.87×10^3	6.65×10^3
	podas	8.23×10^3	8.13×10^3	8.14×10^3	6.02×10^3	6.65×10^3
300	tiempo	1166	588	319	1176	300
	cheq	1.95×10^7	5.40×10^6	2.52×10^6	2.83×10^6	1.83×10^6
	prop	9.49×10^3	2.61×10^3	2.61×10^3	6.97×10^3	6.29×10^3
	podas	9.28×10^3	9.05×10^3	9.05×10^3	7.13×10^3	6.29×10^3
400	tiempo	1297	616	371	1536	408
	cheq	2.05×10^7	5.00×10^6	2.77×10^6	3.66×10^6	2.52×10^6
	prop	9.71×10^3	2.69×10^3	2.68×10^3	6.67×10^3	5.92×10^3
	podas	9.52×10^3	9.26×10^3	9.27×10^3	6.82×10^3	5.92×10^3
500	tiempo	1526	677	437	1875	471
	cheq	2.23×10^7	5.11×10^6	3.07×10^6	4.40×10^6	2.99×10^6
	prop	1.16×10^4	2.88×10^3	2.88×10^3	6.91×10^3	6.20×10^3
	podas	9.71×10^3	9.32×10^3	9.32×10^3	7.06×10^3	6.20×10^3
600	tiempo	1593	695	484	2207	537
	cheq	2.13×10^7	4.84×10^6	3.25×10^6	5.07×10^6	3.38×10^6
	prop	1.18×10^4	2.75×10^3	2.75×10^3	7.29×10^3	6.33×10^3
	podas	9.72×10^3	9.27×10^3	9.27×10^3	7.44×10^3	6.33×10^3
700	tiempo	1706	750	546	2550	575
	cheq	2.09×10^7	4.89×10^6	3.47×10^6	5.68×10^6	3.76×10^6
	prop	1.21×10^4	2.86×10^3	2.86×10^3	7.51×10^3	6.81×10^3
	podas	9.75×10^3	9.23×10^3	9.23×10^3	7.66×10^3	6.81×10^3
800	tiempo	1795	779	592	2893	647
	cheq	2.04×10^7	4.83×10^6	3.66×10^6	6.22×10^6	4.12×10^6
	prop	1.21×10^4	2.81×10^3	2.81×10^3	7.90×10^3	7.08×10^3
	podas	9.77×10^3	9.13×10^3	9.13×10^3	8.05×10^3	7.09×10^3

Como en la tabla anterior, los resultados fueron similares. Los resultados muestran que 2-C3OPL tuvo mejor comportamiento promedio que el resto de los algoritmos. 2-C3OPL fue más rápido que 2-C3OP (39.24 %), 2-C3 (68 %), 2-C4 (75 %) y 2-C6 (2 %). También, 2-C3OPL realizó menos chequeos de restricciones y menos propagaciones que el resto de algoritmos de 2-consistencia. Comparando los algoritmos de 2-consistencia con los algoritmos de arco-consistencia, se observa que 2-C3OPL realiza menos propagaciones que AC3-NN y AC7 (en un orden de magnitud); AC4, AC4-OP y AC6 (50 % menos). Aunque AC3-NN fue un 30 % más rápido que 2-C3OPL, esto no garantiza que detecte la inconsistencia en problemas no-normalizados, por lo que consideramos que 2-C3OPL es la mejor elección en esta tipología de problemas.

6.2. Problemas benchmarks

En esta sección vamos a analizar el comportamiento de los algoritmos desarrollados sobre problemas existentes en la literatura.

6.2.1. Problema de las palomas

El problema de las palomas (del inglés Pigeons problem)[1] es un ejemplo bien conocido en la comunidad de CSP de problemas sin solución. El problema consiste en colocar n palomas en un palomar que tiene $n-1$ casilleros, donde cada casillero del palomar admite únicamente a una paloma. El problema puede ser formulado como un CSP con n variables (correspondientes a las n palomas), donde cada variable tiene $n-1$ valores de dominio (correspondiente a los casilleros) y cada variable está restringida con el resto de variables en el problema. De esta forma, todas las restricciones son binarias y todas las variables tienen la misma talla de dominio. Hay dos tipos de problemas Pigeons: normalizados y no-normalizados. Se escogieron los problemas no-normalizados para realizar la evaluación. Las instancias originales no-normalizadas de este problema benchmark tienen dos restricciones entre cada par de variables: $\forall i < j : X_i \leq X_j$ y $X_i \neq X_j$, por lo que son problemas 100 % no-normalizados. La composición de cada instancia del problema Pigeons ⟨variables,

[1] Los problemas benchmark Pigeons están disponibles en http://www.cril.univ-artois.fr/CPAI08/

dominios y restricciones⟩ es mostrada en la Tabla 6.18.

Tabla 6.18: Datos de los problemas Pigeons.

instancia	**Variables**	**Dominios**	**Restricciones**
10	10	0..8	90
20	20	0..18	380
30	30	0..28	870
40	40	0..38	1560
50	50	0..48	2450

Evaluación de técnicas de arco-consistencia

La Tabla 6.19 muestra el comportamiento de AC6, AC7, AC4, AC4-OPNN y AC3-NN en diferentes instancias del problema Pigeons. Las combinaciones de variables, dominios y restricciones utilizadas se muestran en la Tabla 6.18.

Los resultados indican que todas las instancias son arco-consistentes, a pesar de que los problemas no tienen solución. Sin embargo, esto no es detectado por ninguno de los algoritmos de arco-consistencia, que no realizaron ninguna poda y por lo tanto tampoco realizan propagaciones. Analizando individualmente las técnicas de arco-consistencia para estas instancias, AC3-NN tuvo el mejor tiempo de cómputo (9.4 milisegundos en promedio), incluso por debajo de AC6 (37.4 milisegundos en promedio); AC4-OPNN -pese a buscar todos los soportes- fue más rápido que AC7 y AC4 en un 95 % y 3.27 %, respectivamente; y AC7 fue el más lento (tardó 17153 milisegundos en promedio). El tiempo de cómputo entre AC4 y AC4-OPNN fue similar debido a la baja restringibilidad de los problemas. AC7 fue el que realizó menor cantidad de chequeos de restricciones (82985, en promedio), AC3-NN y AC6 realizaron la misma cantidad (532590, en promedio), mientras que AC4-OPNN realizó un 50 % menos chequeos de restricciones que AC4 (AC4-OPNN: 1826270 y AC4: 3652540, en promedio), aunque ambos buscan todos los soportes, AC4-OPNN realiza el proceso bidireccionalmente.

Tabla 6.19: Resultados de las técnicas de arco-consistencia AC6, AC7, AC4, AC4-OPNN y AC3-NN en las instancias Pigeons mostradas en la Tabla 6.18

instancia	podas		AC6	AC7	AC4	AC4-OPNN	AC3-NN
10	0	tiempo	0	0	0	0	0
		cheq	3.33×10^{03}	1.40×10^{03}	1.46×10^{04}	7.29×10^{03}	3.33×10^{03}
		soportes	10	10	6.57×10^{03}	6.57×10^{03}	10
20	0	tiempo	0	562	63	62	0
		cheq	4.73×10^{04}	1.35×10^{04}	2.74×10^{05}	1.37×10^{05}	4.73×10^{04}
		soportes	20	20	1.30×10^{05}	1.30×10^{05}	20
30	0	tiempo	15	4562	360	344	0
		cheq	2.28×10^{05}	4.83×10^{04}	1.46×10^{06}	7.32×10^{05}	2.28×10^{05}
		soportes	30	30	7.07×10^{05}	7.07×10^{05}	30
40	0	tiempo	47	19782	1156	1109	16
		cheq	7.01×10^{05}	1.18×10^{05}	4.75×10^{06}	2.37×10^{06}	7.01×10^{05}
		soportes	40	40	2.31×10^{06}	2.31×10^{06}	40
50	0	tiempo	125	60860	2860	2781	31
		cheq	1.68×10^{06}	2.34×10^{05}	1.18×10^{07}	5.88×10^{06}	1.68×10^{06}
		soportes	50	50	5.76×10^{06}	5.76×10^{06}	50
Todos los algoritmos indican que todas las instancias son arco-consistentes							

Evaluación de técnicas de 2-consistencia

La Tabla 6.20 muestra el comportamiento de AC3NH, 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6 en las instancias que se muestran en la Tabla 6.18. Los resultados indican que todas las instancias son inconsistentes, contrariamente a los resultados obtenidos por los algoritmos de arco-consistencia (ver Tabla 6.19).

Los resultados indican que todos los algoritmos de 2-consistencia realizaron la misma cantidad de podas pero diferente cantidad de propagaciones. Los tiempos de cómputo promedio (en milisegundos) ordenados en forma ascendente fueron los siguientes: 2-C6 (65.8); 2-C3 (121.8); 2-C3OP (156.2); 2-C3OPL (159.4); 2-C4 (231.2) y AC3NH (7881.4). Por lo tanto 2-C6 fue el más rápido. En cuanto a la cantidad de conjuntos de restricciones evaluados, los algoritmos 2-C3OP, 2-C3OPL y 2-C4 evaluaron la misma cantidad de conjuntos (535 en promedio), lo que equivale a un 50 % menos que los conjuntos evaluados por 2-C3 y 2-C6 (ambos realizaron 1069 evaluaciones en promedio). En cuanto a la cantidad de chequeos de restricciones, 2-C3OPL y 2-C3OP realizaron la menor y misma cantidad de chequeos en promedio (467781), seguidos por 2-C6 (522927), 2-C3 (791632), 2-C4 (892761) y AC3NH (1826270). La diferencia de tiempo y chequeos realizada por AC3NH se debe al proceso de normalización.

Comparando las técnicas de 2-consistencia con las técnicas de arco-consistencia, los resultados indican que las técnicas de 2-consistencia realizaron en promedio 563 podas, mientras que las técnicas de arco-consistencia no realizaron ninguna poda. Las técnicas de 2-consistencia requirieron menos tiempo que algunas de las técnicas propuestas en la literatura: por ejemplo, en la instancia 50, en la que realizó 1270 podas, 2-C6 requirió el doble del tiempo que AC6, un 93 % menos de tiempo que AC4 y 99 % menos de tiempo que AC7. En comparación con AC3-NN, 2-C6 tarda 86 % más tiempo. Pero AC4, AC6, AC7 y AC3-NN no hicieron ninguna poda, ni detectaron la inconsistencia. Así, el tiempo empleado por las técnicas de arco-consistencia es tiempo perdido, ya que traslada el problema a las técnicas de búsqueda, por lo que en este tipo de problemas, las técnicas de 2-consistencia son la mejor elección, encabezadas por 2-C6 y seguidas por 2-C3OPL.

Tabla 6.20: Resultados de las técnicas de consistencia AC3NH, 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6 en las instancias Pigeons mostradas en la Tabla 6.18

instancia	podas		AC3NH	2-C3	2-C3OP	2-C3OPL	2-C4	2-C6
10	53	tiempo	0	0	0	0	0	0
		SetChecks	-	89	45	45	45	89
		cheq	7.29×10^{03}	4.61×10^{03}	2.06×10^{03}	2.06×10^{03}	3.32×10^{03}	2.98×10^{03}
		prop	1.13×10^{04}	42	63	63	42	53
20	208	tiempo	125	0	0	15	15	0
		SetChecks	-	3.79×10^{02}	1.90×10^{02}	1.90×10^{02}	1.90×10^{02}	3.79×10^{02}
		cheq	1.37×10^{05}	6.90×10^{04}	3.63×10^{04}	3.63×10^{04}	6.53×10^{04}	4.52×10^{04}
		prop	4.40×10^{05}	187	323	323	187	208
30	463	tiempo	1282	46	47	47	94	32
		SetChecks	-	8.69×10^{02}	4.35×10^{02}	4.35×10^{02}	4.35×10^{02}	8.69×10^{02}
		cheq	7.32×10^{05}	3.37×10^{05}	1.90×10^{05}	1.90×10^{05}	3.54×10^{05}	2.22×10^{05}
		prop	3.57×10^{06}	432	783	783	432	463
40	818	tiempo	7719	156	171	172	297	94
		SetChecks	-	1.56×10^{03}	7.80×10^{02}	7.80×10^{02}	7.80×10^{02}	1.56×10^{03}
		cheq	2.37×10^{06}	1.04×10^{06}	6.09×10^{05}	6.09×10^{05}	1.16×10^{06}	6.87×10^{05}
		prop	1.55×10^{07}	7.77×10^{02}	1.44×10^{03}	1.44×10^{03}	7.77×10^{02}	8.18×10^{02}
50	1270	tiempo	30281	407	563	563	750	203
		SetChecks	-	2.45×10^{03}	1.23×10^{03}	1.23×10^{03}	1.23×10^{03}	2.45×10^{03}
		cheq	5.88×10^{06}	2.51×10^{06}	1.50×10^{06}	1.50×10^{06}	2.88×10^{06}	1.66×10^{06}
		prop	4.83×10^{07}	1.22×10^{03}	2.30×10^{03}	2.30×10^{03}	1.22×10^{03}	1.27×10^{03}
Todos los algoritmos indican que todas las instancias son inconsistentes								

6.2.2. Extensión de los problemas Pigeons

A las instancias de los problemas Pigeons mostradas en la Tabla 6.18, se les realizaron las siguientes extensiones: le aumentamos el número de casilleros (se incrementó la talla del dominio de cada problema) e incrementamos la cantidad de palomas (lo que significa, un incremento en la cantidad de variables y restricciones), con la finalidad de obtener instancias consistentes (arco-consistentes/2-consistentes) con solución, 100 % no-normalizadas. Todas las instancias continúan con las mismas dos restricciones entre cada par de variables: $\forall i < j : X_i \leq X_j$ y $X_i \neq X_j$. A estas instancias las denominamos problemas Pigeons con solución, y se les aplicará un algoritmo de consistencia y posteriormente un algoritmo de búsqueda (para encontrar una solución). Así mismo, utilizando estos problemas, evaluaremos la capacidad de procesamiento de las técnicas de arco-consistencia y 2-consistencia, tanto de la literatura como las propuestas en esta tesis. Las combinaciones de variables, dominios y restricciones utilizadas en las instancias Pigeons con solución evaluadas son mostradas en la Tabla 6.21.

Tabla 6.21: Datos de los problemas Pigeons con solución.

instancia	**Variables**	**Dominios**	**Restricciones**
10	10	0..15	90
20	20	0..25	380
30	30	0..40	870
40	40	0..45	1560
50	50	0..55	2450
60	60	0..65	3540
70	70	0..75	4830
80	80	0..85	6320
100	100	0..120	9900
200	200	0..220	39800
300	300	0..320	89700
400	400	0..420	159600
500	500	0..520	249500

Evaluación de técnicas de arco-consistencia

La Tabla 6.22 muestra el tiempo de cómputo en milisegundos (tiempo), el número de chequeos de restricciones (cheq) y el número de soportes encontrados (soportes) en las instancias Pigeons con solución, al aplicar técnicas de arco-consistencia donde el número de variables fue incrementado de 50 a 500, la talla del dominio incrementada de 55 a 520 y el número de restricciones fue incrementado de 2450 a 249500.

Los resultados muestran que:

- los algoritmos de arco-consistencia de grano-fino no pudieron procesar todas las instancias. Así, AC6 procesó hasta la instancia 300; AC7 hasta la instancia 200; AC4 y AC4-OPNN hasta la instancia 80. Sin embargo AC3-NN (arco-consistencia de grano-grueso) si pudo realizar el procesamiento de todas las instancias. Esto es debido a que los algoritmos de grano fino tienen más coste espacial que los de grano-grueso;

- ninguno de los algoritmos de arco-consistencia realizaron podas y por lo tanto tampoco realizaron propagaciones. Esto es debido a que al analizar individualmente las restricciones con operadores $\leq$ y $\neq$, y que todas las variables tienen la misma talla de dominios, todos los valores encuentran *soportes* (cosa que no pasaría si las restricciones se analizaran en conjuntos de restricciones);

- en cuanto al tiempo de cómputo, el algoritmo AC3-NN fue el más rápido, seguido por AC6 y AC4-OPNN. El más lento fue AC7 (tres órdenes de magnitud más lento que AC3-NN). AC4-OPNN fue un 4 % más veloz que AC4;

- en cuanto al número de chequeos, AC7 fue el algoritmo que realizó menor cantidad de chequeos (por ejemplo, en la instancia 80 AC7 realizó un 6 % menos chequeos que AC6 y AC3-NN); AC6 y AC3-NN realizaron la misma cantidad de chequeos y AC4-OPNN realizó un 50 % menos de chequeos que AC4;

- la cantidad de soportes que consiguen los algoritmos AC4 y AC4-OPNN es superior en dos órdenes de magnitud en relación a AC6, AC7 y AC3-NN.

Por las razones arriba indicadas, AC3-NN es la mejor opción de los algoritmos de arco-consistencia en este tipo de instancias.

Tabla 6.22: Resultados de las técnicas de arco-consistencia AC6, AC7, AC4, AC4-OPNN y AC3-NN en las instancias Pigeons con solución mostradas en la Tabla 6.21

instancia	podas		AC6	AC7	AC4	AC4-OPNN	AC3-NN
50	0	tiempo	141	70032	3812	3672	31
		cheq	2.16×10^{6}	2.68×10^{5}	1.54×10^{7}	7.68×10^{6}	2.16×10^{06}
		soportes	50	50	7.55×10^{6}	7.55×10^{6}	50
80	0	tiempo	593	177377	24797	23875	187
		cheq	1.26×10^{7}	1.18×10^{7}	9.35×10^{7}	4.67×10^{7}	1.26×10^{07}
		soportes	80	80	4.62×10^{7}	4.62×10^{7}	80
100	0	tiempo	1469	624789	NP	NP	563
		cheq	3.83×10^{7}	3.65×10^{7}	NP	NP	3.83×10^{07}
		soportes	100	100	NP	NP	100
200	0	tiempo	13875	24455423	NP	NP	7125
		cheq	5.01×10^{8}	4.88×10^{8}	NP	NP	5.01×10^{08}
		soportes	200	200	NP	NP	200
300	0	tiempo	56485	NP	NP	NP	33125
		cheq	2.36×10^{9}	NP	NP	NP	2.36×10^{09}
		soportes	300	NP	NP	NP	300
400	0	tiempo	NP	NP	NP	NP	100626
		cheq	NP	NP	NP	NP	7.19×10^{09}
		soportes	NP	NP	NP	NP	400
500	0	tiempo	NP	NP	NP	NP	239424
		cheq	NP	NP	NP	NP	1.72×10^{10}
		soportes	NP	NP	NP	NP	500

Evaluación de técnicas de 2-consistencia

La Tabla 6.23 muestra el tiempo de cómputo en milisegundos (tiempo), el número de conjuntos (SetChecks), el número de chequeos de restricciones (cheq), el número de propagaciones (prop), el número de podas (podas) y el número de soportes encontrados (soportes) en las instancias Pigeons con solución, al aplicar técnicas de 2-consistencia sobre las instancias anteriormente evaluadas.

Los resultados muestran que:

- los algoritmos de 2-consistencia de grano-fino (2-C4 y 2-C6) no pudieron procesar todas las instancias, mientras que los de grano-grueso si pudieron. En este sentido, 2-C6 procesó hasta la instancia 400; 2-C4 procesó hasta la instancia 100; AC3NH fue el que menos instancias procesó (sólo procesó hasta la instancia 90); mientras que 2-C3, 2-C3OP y 2-C3OPL procesaron todas las instancias. Comparándolo con sus algoritmos de arco-consistencia base, los algoritmos de 2-consistencia pudieron procesar una instancia más. Esto es debido a que los algoritmos de 2-consistencia de grano fino tienen menos coste espacial que los algoritmos de arco-consistencia de grano-fino;

- todos los algoritmos de 2-consistencia realizaron la misma cantidad de podas y esta se ubica de 3 a 5 grados de magnitud sobre los algoritmos de arco-consistencia (que no realizaron poda);

- en cuanto al tiempo de cómputo, el algoritmo 2-C6 fue el más rápido, seguido por 2-C4 (ambos hasta las instancias que pudieron procesar). El más lento fue AC3NH (dos órdenes de magnitud más lento que AC6). Hasta la instancia 200, 2-C3OPL fue un 27 % más rápido que 2-C3 y un 40 % más rápido que 2-C3OP. A partir de la instancia 300, 2-C3 fue más rápido en un 30 % que 2-C3OPL y en un 33 % que 2-C3OP, lo cual se debe la bidireccionalidad y a que evitar chequeos es rentable hasta cierta talla de problemas. Después las estructuras que soportan la bidireccionalidad se tornan costosas. El tiempo empleado por los algoritmos de 2-consistencia fue inferior que el empleado por AC7 (ver instancia 200 en la Tabla 6.22);

- el número de conjunto de restricciones procesados fue un orden de magnitud menor en 2-C4 que en el resto de algoritmos; le siguen en orden descendente

2-C6, 2-C3OP y 2-C3OPL, estos dos últimos con igual cantidad de conjuntos procesados. En este caso al haber propagaciones 2-C6 se beneficia de procesar nuevamente sólo los valores que han sufrido del dominio de la variable, a diferencia de 2-C3OP y 2-C3OPL que deben procesar el dominio completo. 2-C3 fue el que procesó mayor cantidad de conjuntos de restricciones.

- en cuanto al número de chequeos, 2-C3OPL fue el algoritmo que realizó menor cantidad de chequeos. Así, por ejemplo, en la instancia 80, 2-C3OPL realizó un 5 % menos que 2-C6; un orden de magnitud menos que 2-C3OP y AC3NH y dos órdenes de magnitud menos que 2-C3.

- en cuanto al número de propagaciones, 2-C6 realizó menos cantidad de propagaciones: dos órdenes de magnitud menos que 2-C3, 2-C3OP y 2-C3OPL y cinco órdenes de magnitud menos en relación a AC3NH;

- la cantidad de soportes que consigue el algoritmo 2-C4 fue superior en dos órdenes de magnitud en relación a 2-C3, 2-C3OP, 2-C3OPL, 2-C6 y AC3NH.

Por las razones arriba indicadas, los algoritmos de 2-consistencia son los más apropiados en este tipo de problema. Así, hasta la instancia 400, 2-C6 fue la mejor opción de los algoritmos de 2-consistencia y a partir de la instancia 500 lo fue el algoritmo 2-C3.

Tabla 6.23: Resultados de las técnicas de 2-consistencia AC3NH, 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6 en las instancias Pigeons con solución mostradas en la Tabla 6.21

instancia	podas		AC3NH	2-C3	2-C3OP	2-C3OPL	2-C4	2-C6
50	2.45×10^{03}	tiempo	53641	2360	2844	1719	1234	312
		SetChecks	-	6.01×10^{04}	6.00×10^{04}	6.00×10^{04}	1.23×10^{03}	1.09×10^{04}
		cheq	7.68×10^{06}	1.88×10^{07}	1.29×10^{07}	2.18×10^{06}	4.11×10^{06}	2.33×10^{06}
		prop	4.33×10^{08}	5.77×10^{04}	5.88×10^{04}	5.88×10^{04}	2.40×10^{03}	2.45×10^{03}
		soportes	50	50	50	50	1.15×10^{05}	50
80	6.32×10^{03}	tiempo	745229	18282	22281	16610	7047	1172
		SetChecks	-	2.50×10^{05}	2.50×10^{05}	2.50×10^{05}	3.16×10^{03}	2.94×10^{04}
		cheq	4.67×10^{07}	1.09×10^{08}	6.07×10^{07}	6.91×10^{06}	1.86×10^{07}	7.31×10^{06}
		prop	4.96×10^{09}	2.43×10^{05}	2.46×10^{05}	2.46×10^{05}	6.24×10^{03}	6.32×10^{03}
		soportes	80	80	80	80	3.01×10^{05}	80
100	9.90×10^{03}	tiempo	NP	67251	79095	51672	23141	4016
		SetChecks	NP	4.90×10^{05}	4.90×10^{05}	4.90×10^{05}	4.95×10^{03}	1.00×10^{05}
		cheq	NP	5.17×10^{08}	2.81×10^{08}	2.23×10^{07}	6.18×10^{07}	2.32×10^{07}
		prop	NP	4.80×10^{05}	4.85×10^{05}	4.85×10^{05}	9.80×10^{03}	9.90×10^{03}
		soportes	NP	100	100	100	4.46×10^{06}	100
200	3.98×10^{04}	tiempo	NP	1930634	2273701	1758442	NP	54329
		SetChecks	NP	3.96×10^{06}	3.96×10^{06}	3.96×10^{06}	NP	4.49×10^{05}
		cheq	NP	1.00×10^{10}	5.28×10^{09}	2.72×10^{08}	NP	2.79×10^{08}
		prop	NP	3.92×10^{06}	3.94×10^{06}	3.94×10^{06}	NP	3.98×10^{04}
		soportes	NP	200	200	200	NP	200
400	1.60×10^{05}	tiempo	NP	64958941	94936448	80936458	NP	921433
		SetChecks	NP	3.18×10^{07}	3.18×10^{07}	3.18×10^{07}	NP	1.90×10^{06}
		cheq	NP	2.38×10^{11}	1.23×10^{11}	3.75×10^{09}	NP	3.80×10^{09}
		prop	NP	3.17×10^{07}	3.18×10^{07}	3.18×10^{07}	NP	1.60×10^{05}
		soportes	NP	400	400	400	NP	400
500	2.50×10^{05}	tiempo	NP	329795840	491917474	470245648	NP	NP
		SetChecks	NP	6.23×10^{07}	6.22×10^{07}	6.22×10^{07}	NP	NP
		cheq	NP	6.81×10^{11}	3.50×10^{11}	8.88×10^{09}	NP	NP
		prop	NP	6.20×10^{07}	6.21×10^{07}	6.21×10^{07}	NP	NP
		soportes	NP	500	500	500	NP	NP

Evaluación de las técnicas de búsqueda

Para la evaluación de las técnicas de búsqueda se utilizaron las mismas instancias Pigeons con solución mostradas en la Tabla 6.21 donde aplicamos técnicas de arco-consistencia y 2-consistencia. Pese a que desarrollamos dos algoritmos de búsqueda: BLS y SchTrains, el algoritmo SchTrains sólo es aplicable al dominio de planificación de horarios ferroviarios -que será evaluado en el próximo capítulo-, por lo que en este apartado sólo compararemos el algoritmo de búsqueda BLS con los algoritmos de búsqueda señalados en el Capítulo 2 (BT, FC y RFLA). Además, se considera que previo a la búsqueda se ha aplicado la técnica de consistencia más efectiva para este tipo de problemas.

- **Evaluación de técnicas de búsqueda BT y BLS**

La Figura 6.1 muestra el tiempo de cómputo empleado por los algoritmos BT y BLS en encontrar una solución en diferentes instancias Pigeons con solución (instancias 30 hasta 100, mostradas en la Tabla 6.21), cuando se aplica una técnica de arco-consistencia y una técnica de búsqueda. En este sentido, el algoritmo de BT se ejecutó posteriormente de haber realizado el proceso de arco-consistencia (en etapa de pre-proceso) con los algoritmos AC4, AC6, AC7 y AC3-NN.

Los resultados indican que el algoritmo BLS tuvo un comportamiento intermedio, ya que BLS fue más rápido que utilizar BT con AC4 y BT con AC7, pero no lo fue cuando se utilizó BT con AC6 y BT con AC3-NN. El algoritmo BLS realiza la búsqueda en menor tiempo que BT, pero debe sumar el tiempo que el algoritmo 2-C3OPL consume al realizar el proceso de 2-consistencia, lo cual hace que un algoritmo *poco eficiente* como el BT saque provecho y mejor tiempo de cómputo. Cabe destacar que en estas instancias, algunos de los algoritmos de arco-consistencia no pudieron procesar todas las instancias. En este sentido, AC4 pudo procesar hasta la instancia 80 y AC6 pudo procesar hasta la instancia 300, por lo que la mejor opción en estas instancias entre BLS y BT, fue la utilización de BT con AC3-NN.

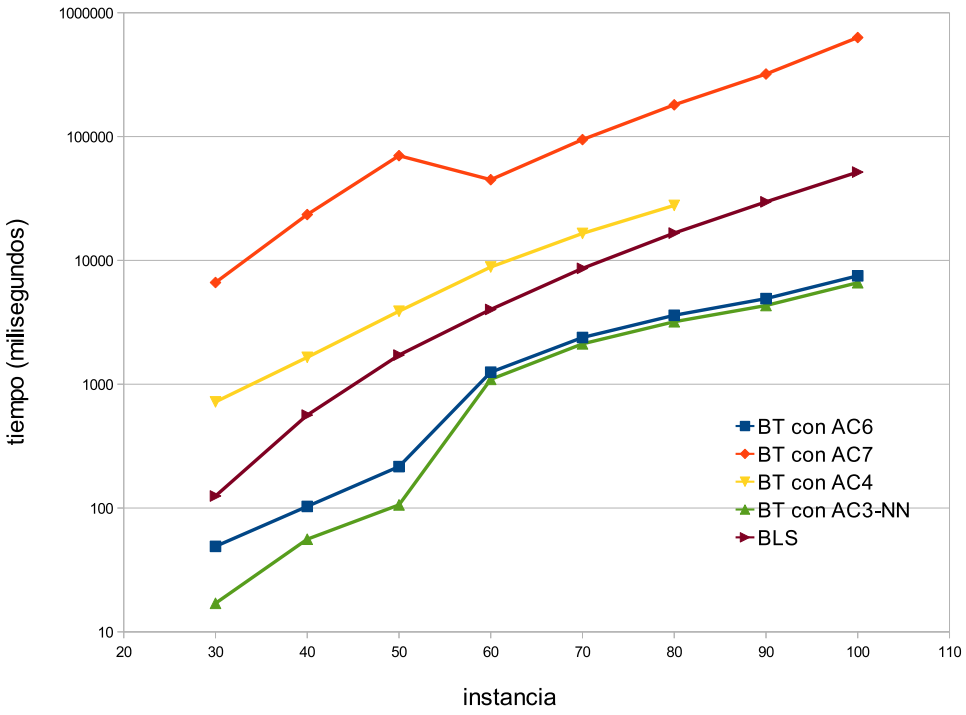


Figura 6.1: Tiempo de cómputo (milisegundos) utilizado por los algoritmos de búsqueda BLS y BT (combinado con las técnicas de arco-consistencia: AC4, AC6, AC7 y AC3-NN) en instancias Pigeons con solución mostradas en la Tabla 6.21.

- **Evaluación de técnicas de búsqueda FC y BLS**

La Figura 6.2 muestra el tiempo de cómputo empleado por los algoritmos FC y BLS en encontrar una solución en diferentes instancias Pigeons con solución (instancias 30 hasta 100, mostradas en la Tabla 6.21). El algoritmo FC se ejecutó posteriormente de haber realizado el proceso de arco-consistencia (en etapa de pre-proceso) con los algoritmos AC4, AC6, AC7 y AC3-NN. Los resultados indican que BLS tuvo mejor comportamiento en todas las instancias comparado con FC ejecutado con los diferentes algoritmos de arco-consistencia, ya que los algoritmos de arco-consistencia en la etapa de pre-proceso sólo consumieron tiempo (no realizan ninguna poda) y FC debe realizar chequeos de consistencia al realizar las instanciaciones. FC con AC3-NN tuvo resultados ligeramente superiores a FC con AC6, lo que hace que se

solapen en la gráfica. Así, la mejor opción en técnicas de búsqueda en estas instancias entre los algoritmos BLS y FC fue la utilización de BLS.

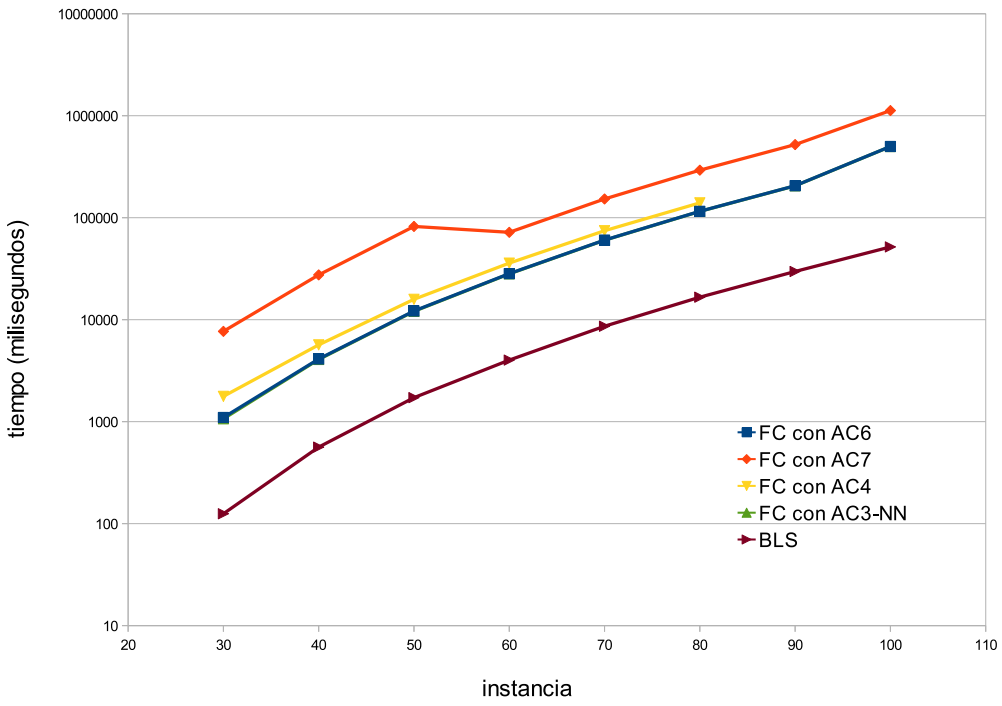


Figura 6.2: Tiempo de cómputo (milisegundos) utilizado por los algoritmos de búsqueda BLS y FC (combinado con las técnicas de arco-consistencia: AC4, AC6, AC7 y AC3-NN) en instancias Pigeons con solución mostradas en la Tabla 6.21.

- **Evaluación de técnicas de búsqueda RFLA y BLS**

La Figura 6.3 muestra el tiempo de cómputo empleado por los algoritmos RFLA y BLS en encontrar una solución en diferentes instancias Pigeons con solución (instancias 30 hasta 100, mostradas en la Tabla 6.21). El algoritmo RFLA se ejecutó posteriormente de haber realizado el proceso de arco-consistencia (en etapa de pre-proceso) con los algoritmos AC4, AC6, AC7 y AC3-NN. Cabe destacar que en este tipo de problemas donde la asignación de valores válidos a las variables es sencilla de realizar, la simplicidad de procesamiento de BT se pone de manifiesto,

resultando ser aún mejor opción que los algoritmos FC y RFLA. Los resultados con RFLA fueron bastantes similares a los obtenidos con FC -con peor desempeño-, por lo que el algoritmo BLS fue mejor opción que RFLA en estas instancias.

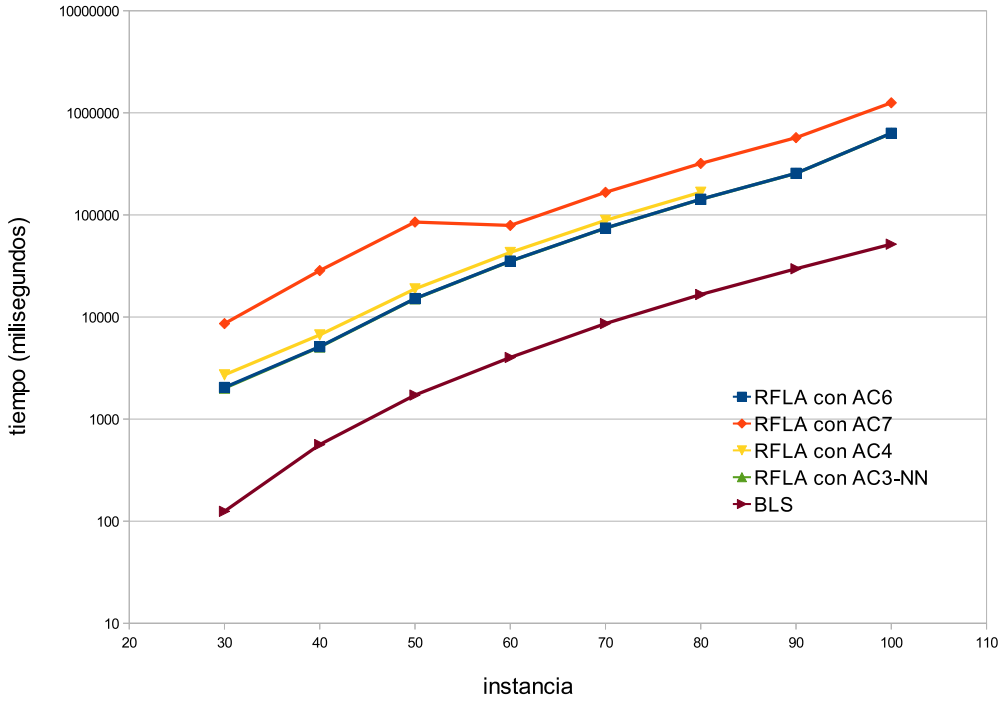


Figura 6.3: Tiempo de cómputo (milisegundos) utilizado por los algoritmos de búsqueda BLS y RFLA (combinado con las técnicas de arco consistencia: AC4, AC6, AC7 y AC3-NN) en instancias Pigeons con solución mostradas en la Tabla 6.21.

- **Evaluación de técnicas de búsqueda BT, FC y RFLA con 2-consistencia**

Cabe destacar que los algoritmos BT, FC y RFLA también se benefician al ser ejecutados con posterioridad a la aplicación de un algoritmo de 2-consistencia como 2-C6. La Figura 6.4 muestra los resultados de comparar BLS con las técnicas de búsqueda BT, FC y RFLA al recibir pre-proceso de las técnicas más rápidas de

arco-consistencia (AC3-NN) y de 2-consistencia (2-C6), sobre las instancias Pigeons con solución mostradas en la Tabla 6.21. BT con 2-C6 tuvo un tiempo ligeramente superior a RFLA con 2-C6, por lo que se solapan en la gráfica. Como puede observarse, hasta la instancia 50, BT al ser aplicado con AC3-NN tuvo menor tiempo de cómputo, pero a partir de la instancia 60, BT tuvo menor tiempo de cómputo al ser aplicado con 2-C6. BLS mantuvo un tiempo de cómputo promedio (mejor que FC y BT con AC3-NN) en todas las instancias e incluso BLS tuvo mejor tiempo de cómputo en instancias pequeñas (hasta la instancia 40) que FC al ser aplicado con 2-C6. Por lo en estas instancias la combinación con 2-C6 y una técnica de búsqueda fue la mejor opción.

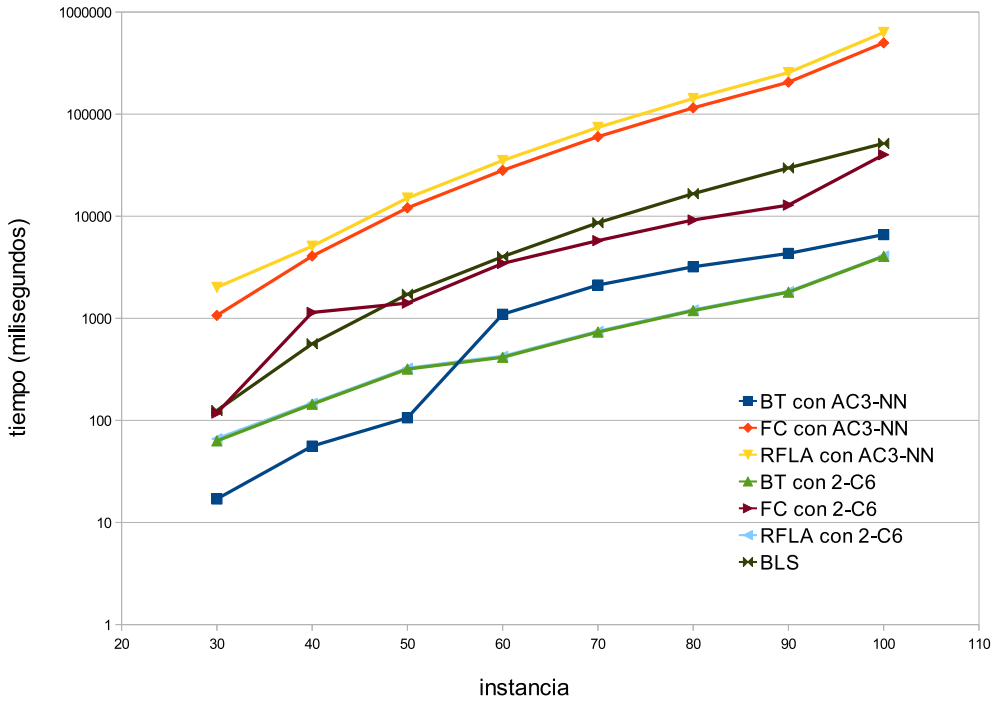


Figura 6.4: Tiempo de cómputo (milisegundos) utilizado por los algoritmos de búsqueda BT, FC, RFLA (combinados con las técnicas de consistencia: AC3-NN y 2-C6) y BLS en instancias Pigeons con solución mostradas en la Tabla 6.21.

6.3. Conclusiones

Proponer algoritmos eficientes que alcancen la arco-consistencia ha sido un punto central en la comunidad que investiga el razonamiento con restricciones. Trabajando sobre los algoritmos de arco-consistencia existentes, hemos evaluado experimentalmente los nuevos algoritmos propuestos que alcanzan la arco-consistencia y la 2-consistencia, con respecto a los algoritmos existentes en la literatura. También hemos evaluado la técnica de búsqueda BLS en relación a las técnicas de búsqueda existentes en la literatura (BT, FC y RFLA).

Las pruebas de consistencia se realizaron en problemas binarios normalizados y no-normalizados; consistentes e inconsistentes; en instancias aleatorias y benchmark, donde los algoritmos propuestos lograron mejorar a los algoritmos existentes en la literatura, en dos o más parámetros (número de podas, chequeos de restricciones, número de propagaciones y tiempo de cómputo). Además, demostramos que el proceso de normalización en problemas binarios no-normalizados no tiene un coste trivial, por lo que se demuestra que trabajar con algoritmos de 2-consistencia que procesen dichos problemas no-normalizados sin cambiar el problema original, es una buena elección.

Las pruebas de búsqueda en instancias Pigeons con solución, al ser combinadas con técnicas de arco-consistencia en etapa de pre-proceso, indican que BLS fue más eficiente que las técnicas de búsqueda: FC y RFLA, en todas las instancias y en algunas instancias en relación a BT. Sin embargo, al utilizar estas técnicas en un problema real como el que se presenta en el próximo capítulo, BLS fue superior a BT, FC y RFLA.

Capítulo 7

Aplicación a Problemas Reales

Introducción

En los últimos años, el transporte ferroviario ha tenido un rol importante en muchos países. El tráfico ferroviario se ha incrementado considerablemente[1], lo cual crea la necesidad de optimizar el uso de la infraestructura ferroviaria y los métodos y herramientas para su administración. En este sentido, uno de los objetivos en Europa para el 2020, es lograr un incremento del 20 % en el transporte de pasajeros y del 70 % en mercancías [105]; todo lo cual implica la necesidad de mejorar la gestión de la capacidad en la infraestructura ferroviaria. Un horario de trenes flexible debe especificar los tiempos de salida y llegada a cada estación del trayecto, teniendo en cuenta la capacidad de la línea y otras restricciones operacionales. La construcción de horarios ferroviarios es una tarea difícil que consume mucho tiempo, particularmente en el caso de redes ferroviarias reales, donde el número de restricciones y la complejidad de las mismas crece drásticamente. Motivado por lo anteriormente planteado, se han desarrollado numerosas aproximaciones y herramientas para planificar los horarios ferroviarios. A este tipo de problemas se les conoce como el problema de asignación de horarios ferroviarios [21].

El escenario del problema de asignación de horarios ferroviarios consiste en un conjunto ordenado de dependencias (estaciones, apeaderos, cargaderos, bifurcaciones, etc.), un conjunto de trenes en cada sentido (ida, vuelta) y un recorrido (es el orden en el que el tren visita cada dependencia) para cada tren. El problema

[1] Véanse estadísticas en http://epp.eurostat.ec.europa.eu y http://www.ine.es

trata de asignar los instantes de entrada/salida de los k trenes en las i dependencias minimizando el tiempo promedio de recorrido y satisfaciendo las restricciones del usuario (hora de salida inicial, frecuencia de salida, paradas comerciales, etc.), las restricciones de tráfico (cruce, tiempo de recepción, tiempo de expedición, precedencia, etc.) y las restricciones topológicas (número de vías entre las dependencias, capacidad de cada estación, tiempos de cierre de las estaciones, etc.). Así, el problema de asignación de horarios ferroviarios se caracteriza por poseer un gran número de variables y restricciones.

El problema de asignación de horarios ferroviarios ha sido modelado como un caso especial del problema de asignación de tareas (job-shop scheduling problem) [72, 122, 114], donde los recorridos del tren se consideran los trabajos programados y las vías se consideran los recursos. Así mismo, la asignación de horarios para los nuevos trenes se puede hacer de dos formas: sobre una red 'vacía' (donde no hay trenes previamente planificados) [122, 114] o teniendo en cuenta que la línea de ferrocarril puede estar ocupada por otros trenes en circulación, cuyos horarios no se pueden modificar [72]. En algunos casos, este problema se ha modelado como un CSP y ha sido resuelto utilizando técnicas de programación de restricciones.

La formulación CSP que describimos en este trabajo se ha definido en colaboración con la Administración de Infraestructuras Ferroviarias Española (ADIF), de tal manera que los horarios ferroviarios resultantes sean factibles y viables. La formulación CSP contiene restricciones binarias no-normalizadas, de modo que la mayoría de las técnicas CSP existentes en la literatura no son capaces de gestionar apropiadamente este tipo de restricciones. Aunque un CSP no-normalizado se puede transformar en uno normalizado [14], hacerlo es una tarea muy difícil [7]. Con este fin, se deben desarrollar nuevas técnicas para reducir eficientemente tanto el espacio de búsqueda como el tiempo de cómputo del problema.

7.1. Problema de asignación de horarios ferroviarios

El objetivo principal del problema de asignación de horarios ferroviarios es minimizar el tiempo de recorrido de un conjunto de trenes. Una *red ferroviaria* básicamente está compuesta por dependencias y vías únicas o dobles. Los tipos de *dependencias* que se consideraron son las siguientes:

- *Estación*: es un lugar donde los trenes pueden estacionarse, detenerse o circular. En las estaciones hay dos o más vías donde pueden realizarse cruces y adelantamientos. Cada estación es asociada con un identificador de estación único.
- *Apeadero*: es un lugar donde los trenes pueden detenerse o circular pero no pueden estacionarse, ni gestionar cruces o adelantamientos. Cada apeadero tiene asociado un identificador único.
- *Bifurcación*: es un lugar donde una única vía se divide en dos, o viceversa. No está permitida la parada en la bifurcación. Cada bifurcación está asociada con un identificador único.

En una red ferroviaria, el operador necesita programar los trayectos de n trenes que van en un sentido (ida) y de m trenes que van en dirección opuesta (vuelta). Los trenes pueden ser de diferentes tipos (cercanías, mercancías, regionales, larga distancia, etc.) y cada uno de ellos puede requerir una frecuencia de salida particular. El tipo de tren determina el tiempo necesario para recorrer dos dependencias del trayecto.

El trayecto seleccionado por el operador para el recorrido del tren determina por que estaciones se ha de pasar y los tiempos de parada requeridos en cada estación para propósitos comerciales. Para poder realizar cruces en una sección de vía única, uno de los trenes debe esperar en una estación previa. A esta espera se le denomina parada técnica, donde uno de los trenes es desviado de la vía principal de forma que el otro tren pueda detenerse o continuar.

Los planificadores utilizan los mapas de recorridos como herramientas gráficas que les ayudan en el proceso de planificación. Un *mapa de recorridos* contiene información concerniente a la topología ferroviaria (dependencias, vías, distancia entre dependencias, características de control de tráfico, etc.) y los horarios de los trenes que utilizan la topología ferroviaria (tiempos de llegada y salida de los trenes en cada dependencia, frecuencias, paradas, cruces, adelantamientos, etc.).

La Figura 7.1 muestra un mapa de recorridos donde los nombres de las dependencias son presentados a la izquierda, las líneas verticales representan el número de vías entre dependencias (vía única o vía doble), las líneas punteadas horizontales representan apeaderos o bifurcaciones y las líneas sólidas horizontales representan las

estaciones. El objetivo es obtener un mapa de recorridos válido (que también puede ser optimizado) que tenga en cuenta: (i) las reglas de tráfico, (ii) los requerimientos del usuario y (iii) la topología de la infraestructura ferroviaria.

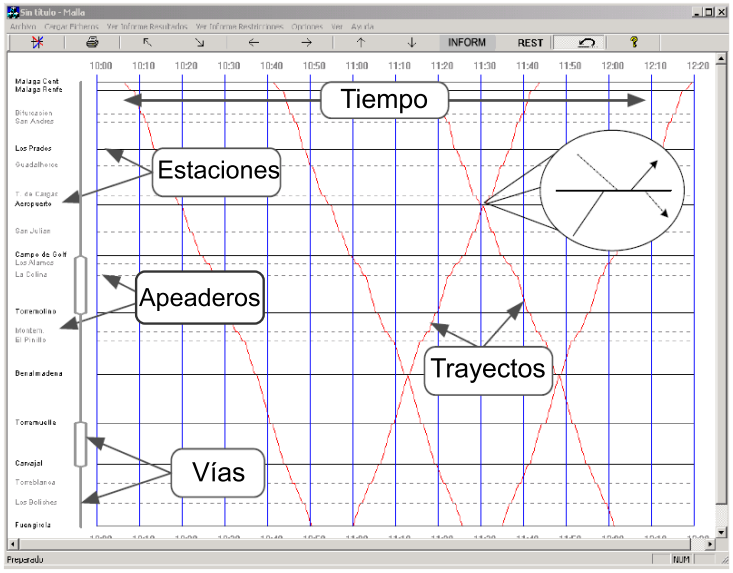


Figura 7.1: Ejemplo de mapa de recorridos para el problema de asignación de horarios ferroviarios.

7.1.1. Notación

La notación utilizada para describir el problema de asignación de horarios ferroviarios está basada en los trabajos de Ingolotti [72] y de Tormos [116], y es la siguiente:

- T: conjunto finito de trenes t considerados en el problema. $T = \{t_1, t_2, ..., t_k\}$. T_D: conjunto de trenes viajando en sentido *ida*. T_U: conjunto de trenes viajando en sentido *vuelta*. De esta forma, $T = T_D \cup T_U$ y $T_D \cap T_U = \emptyset$.

- $L = \{l_0, l_1, ..., l_m\}$: línea ferroviaria compuesta por un conjunto ordenado de dependencias (estaciones, apeaderos y bifurcaciones) que pueden ser visitadas por los trenes $t \in T$. Las dependencias contiguas l_i y l_{i+1} están enlazadas por una sección de vía única o doble.

- J_t: recorrido del tren t. El recorrido se describe como una secuencia ordenada de dependencias a ser visitas por el tren t de forma tal $\forall t \in T, \exists J_t : J_t \subseteq L$. El recorrido J_t muestra el orden utilizado por el tren t para visitar un conjunto de dependencias. De esta forma, l_0^t y $l_{n_t}^t$ representan, respectivamente, la *primera* y *última* dependencia visitada por el tren t.
- C_i^t tiempo mínimo requerido por el tren t para operaciones comerciales (como son embarque y desembarque de pasajeros) en la estación i (parada comercial).
- $\Delta_{i\to(i+1)}^t$: tiempo de recorrido para el tren t desde la dependencia l_i^t a la $l_{(i+1)}^t$.
- F: frecuencia de los trenes consideradas en el problema. F_D: frecuencia de trenes que salen (viajan en sentido *ida*). F_U: frecuencia de los trenes que regresan (viajan en sentido *vuelta*).
- λ: tiempo de retraso permitido para el tren t con una frecuencia F.

7.1.2. Formulación CSP para el problema de asignación de horarios ferroviarios

La formulación CSP para el problema de asignación de horarios ferroviarios que describimos a continuación ha sido definida en colaboración con la Administración de infraestructura ferroviaria española (ADIF), con la finalidad de encontrar una planificación de horarios ferroviario que sea factible y practicable.

Variables

Cada tren cuando pasa por una dependencia generará dos variables diferentes (tiempo de llegada y tiempo de salida):

- dep_i^t tiempo de salida del tren $t \in T$ de la dependencia i, donde $i \in J_t \setminus \{l_{n_t}^t\}$.
- arr_i^t tiempo de llegada del tren $t \in T$ a la dependencia i, donde $i \in J_t \setminus \{l_0^t\}$.

Dominios

El dominio de cada variable (dep_i^t o arr_i^t) es un intervalo $[min_V, max_V]$, donde $min_V \geq 0$ y $max_V > min_V$. Este intervalo es obtenido de ADIF, y puede tener una funcionalidad de segundos o decasegundos.

Restricciones

La asignación de horarios ferroviarios debe satisfacer un conjunto de restricciones para que sea factible. Dichas restricciones las clasificamos en tres grupos, dependiendo de si son: (1) reglas de tráfico, (2) requerimientos del usuario y (3) reglas topológicas. Estas reglas se pueden modelar con las siguientes restricciones:

1. Reglas de Tráfico. Las restricciones tomadas en cuenta son:

   - Restricción de cruce: Dos trenes que viajen en direcciones opuestas no deben utilizar al mismo tiempo la misma sección de vía única: $dep^{t'}_{i+1} > arr^{t}_{i+1} \vee dep^{t}_{i} > arr^{t'}_{i}$, (ver Figura 7.2).

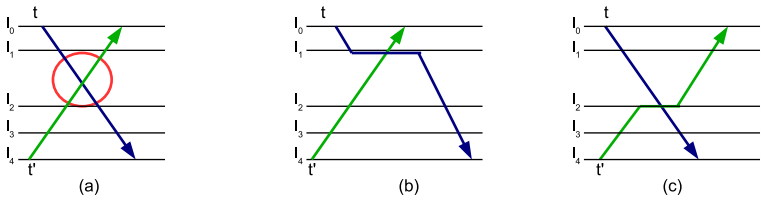


Figura 7.2: Cruce: (a) Conflicto de cruce entre los trenes t y t'. (b) El tren t de ida (de bajada) espera. (c) Tren t' de vuelta (de subida) espera.

   - Restricción de tiempo de expedición. Existe un tiempo dado para colocar un tren desviado de vuelta a la vía principal para que salga de la estación. Así, $|dep^{t'}_{l} - arr^{t}_{l}| \geq E_t$, donde E_t es el tiempo de expedición especificado para el tren t, (ver Figura 7.3).

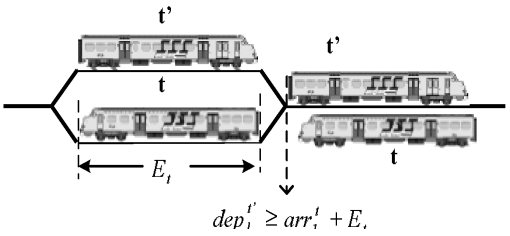


Figura 7.3: Tiempo de expedición entre los trenes t y t'

   - Restricción de tiempo de Recepción. Existe un tiempo dado para desviar un tren desde la vía principal de forma tal que puedan realizarse cruces

y/o adelantamientos: $arr_l^{t'} \geq arr_l^t \rightarrow arr_l^{t'} - arr_l^t \geq Recep_t$, donde $Recep_t$ es el tiempo de recepción especificado para que el tren llegue a l primero, (ver Figura 7.4).

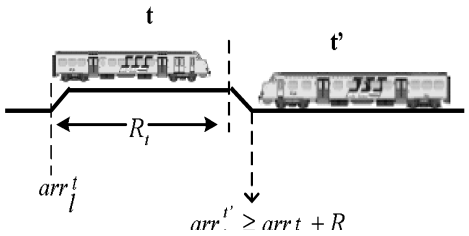


Figura 7.4: Tiempo de Recepción entre los trenes t y t'

- Restricción de tiempo de trayecto: $arr_{i+1}^t = dep_i^t + \Delta_{i\rightarrow(i+1)}^t$. Para cada tren t y cada sección de vías, el tiempo de trayecto está dado por $\Delta_{i\rightarrow(i+1)}^t$, lo cual representa el tiempo que el tren t debe emplear en ir desde la dependencia l_i^t hasta dependencia l_{i+1}^t.

2. Requerimientos del usuario: Las principales restricciones son:

- Número de trenes viajando en cada sentido n (ida) y m (vuelta).
  $T = T_D \cup T_U$, donde:
  - $t \in T_D \leftrightarrow (\forall l_i^t : 0 \leq i < n_t, \exists l_j \in \{L \setminus \{l_m\}\} : l_i^t = l_j \wedge l_{i+1}^t = l_{j+1})$, y
  - $t \in T_U \leftrightarrow (\forall l_i^t : 0 \leq i < n_t, \exists l_j \in \{L \setminus \{l_0\}\} : l_i^t = l_j \wedge l_{i+1}^t = l_{j-1})$.
- Recorrido: dependencias por las que visitar y tiempo de parada para propósitos comerciales en cada sentido para cada tren $t \in T$:
  $J_t = \{l_0^t, l_1^t, ..., l_{n_t}^t\}$
- Horario de frecuencias. Los requerimientos de frecuencia F de la salida de trenes en ambas direcciones: $dep_i^{t+1} - dep_i^t = F + \lambda_i$. Esta restricción es muy restrictiva debido a que cuando son realizados los cruces, los trenes deben esperar en las estaciones un cierto intervalo de tiempo. Este intervalo debe ser propagado a todos los trenes que van en el mismo sentido para mantener el horario de frecuencia establecido. El usuario puede requerir una frecuencia fija, una frecuencia entre un intervalo mínimo y máximo, o múltiples frecuencias.

Una frecuencia entre un intervalo mínimo y máximo fue la seleccionada en esta tesis:

- Para $t \in T_D : dep_i^t + F_D < dep_i^{t+1}$ y $dep_i^t + F_D + \lambda_i > dep_i^{t+1}$.
- Para $t \in T_U : dep_i^t + F_U < dep_i^{t+1}$ y $dep_i^t + F_U + \lambda_i > dep_i^{t+1}$.

3. Topología de la infraestructura ferroviaria. Algunas de estas restricciones son:

   - Número de vías en las estaciones (para realizar operaciones comerciales y paradas técnicas) y el número de vías entre dos dependencias (vía-única o vía-doble). No son permitidos ni los cruces ni los adelantamientos en los tramos de vía única.
   - Restricciones de tiempo adicional en la estación con propósitos comerciales y/o técnicos. Cada tren $t \in T$ está obligado a permanecer en una estación l_i^t por lo menos Com_i^t unidades de tiempo (parada comercial). El resto del tiempo que el tren permanezca en la estación se considerará parada técnica (necesaria para gestionar cruces o adelantamientos): $dep_i^t \geq arr_i^t + Com_i^t$.

De conformidad con los requerimientos de ADIF, el sistema debe obtener una solución para que todas las restricciones antes mencionadas (tráfico, necesidades de los usuarios y topológicas) se cumplan. La fuente de las dificultades subyacente en la asignación de horarios ferroviarios son las siguientes: a) cada dependencia genera dos variables cuyas tallas de dominio son grandes (un problema combinatorio). b) El aumento de los trenes aumenta las restricciones disyuntivas (que genera las ramas del árbol de búsqueda). c) El aumento de la frecuencia de los trenes, hace que el problema sea más restrictivo (a menudo no hay solución). d) Encontrar una solución mediante un algoritmo de búsqueda estándar como FC para las instancias propuestas en esta tésis puede tardar más de doce horas.

7.2. Resultados experimentales

En esta sección, compararemos empíricamente el comportamiento de los algoritmos propuestos en los capítulos anteriores (de consistencia y de búsqueda) con los algoritmos presentes en la literatura para evaluar su rendimiento en el problema de

asignación de horarios ferroviarios. Para los algoritmos de consistencia las medidas de eficiencia consideradas fueron: tiempo de cómputo, cantidad de podas, cantidad de chequeos de restricciones y cantidad de propagaciones. Para los algoritmos de búsqueda la medida de eficiencia fue el tiempo de cómputo. Todos los algoritmos evaluados fueron escritos en C. Los experimentos fueron realizados en un ordenador con procesador Intel Core 2 Quad (de 2.83 GHz velocidad del procesador y 3 GB RAM).

El problema de asignación de horarios ferroviarios que presentamos es una instancia más simple que la propuesta en [72] debido a que asumimos que inicialmente la red ferroviaria está vacía de forma que no haya trenes con horarios previamente asignados en la red, el retraso puede producirse al iniciar el tren su recorrido y que todas las restricciones son binarias.

El CSP generado para el problema de asignación de horarios ferroviarios contiene todas las variables y restricciones anteriormente mencionadas. Las restricciones de cruce fueron generadas de dos formas diferentes: (a) permitiendo todas las posibles combinaciones de cruces para evaluar los algoritmos de consistencia en etapa de pre-proceso; y (b) restringiendo el número de cruces con la heurística 1.0 propuesta por Salido en [108] para generar problemas con solución. Todas las restricciones están presentadas en forma intensional y cada problema posee restricciones no-normalizadas.

Tabla 7.1: Combinación de valores para la frecuencia F y el retraso λ utilizadas en los problemas de asignación de horarios ferroviarios.

Combinación nombre	**F**		retraso λ
	F_D	F_U	
F1	100	120	2
F2	100	120	5
F3	100	120	10
F4	150	170	2
F5	150	170	5
F6	150	170	10
F7	100	150	30

El número de variables y de restricciones en la formulación CSP presentada estuvo determinado por el número de trenes T y el número de dependencias L. Si

L o T se incrementan, entonces tanto el número de variables como el de restricciones se incrementará también. El número de restricciones no cambia cuando se modifican los valores de frecuencia F_D, F_U o el valor de retraso λ, sin embargo esas variaciones influyen en la restringibilidad de las restricciones. Las combinaciones de frecuencias y de trenes utilizadas en la evaluación se muestran en las Tablas 7.1 y 7.2, respectivamente.

Tabla 7.2: Combinación de trenes T utilizadas en los problemas de asignación de horarios ferroviarios.

Combinación nombre	**T**		Número de Trenes
	T_D	T_U	
T0	1	1	2
T1	2	2	4
T2	3	2	5
T3	3	3	6
T4	4	4	8
T5	5	4	9
T6	5	5	10
T7	6	5	11
T8	6	6	12
T9	8	7	15
T10	8	8	16

La evaluación se llevó a cabo en una infraestructura ferroviaria real que une ciudades españolas, a partir de datos facilitados por operadores de ADIF. Consideramos dos recorridos:

- *Zaragoza-Casetas*: Este recorrido consiste en 7 dependencias. En todos estos casos de prueba se fija el número de dependencias (5 estaciones y 2 apeaderos). Sin embargo el número de trenes, la frecuencia y el retraso, se cambian para cada caso de prueba.

- *Zaragoza-Calatayud*: Este recorrido consiste en 25 dependencias. En todos estos casos de pruebas el número de trenes es fijado a 6, sin embargo el número de dependencias (cualquier dependencia intermedia entre Zaragoza y Calatayud), la frecuencia y el retraso son cambiados en cada caso de prueba.

Para cada recorrido (Zaragoza-Casetas y Zaragoza-Calatayud), asignamos los parámetros L, $T_i, (1 \leq i \leq 10)$, y $F_j, (1 \leq j \leq 7)$. Para cada asignación, se obtiene una nueva instancia y su correspondiente CSP $\langle n, d, m, c\rangle$, donde n es el número de variables, d es la talla del dominio, m el número de restricciones binarias y c el número máximo de restricciones en cada conjunto de restricciones. Hay que tener en cuenta que a pesar de que dos instancias diferentes generan la misma parametrización CSP $\langle n, d, m, c\rangle$, ellas representan dos instancias de CSP diferentes debido a la restringibilidad en las restricciones.

7.2.1. Evaluación de técnicas de consistencia

La Tabla 7.3 muestra la cantidad de podas, el tiempo de cómputo y el número de chequeos de restricciones para el problema de asignación de horarios ferroviarios en etapa de pre-proceso en el recorrido *Zaragoza - Casetas* para nueve instancias diferentes. Cada instancia fue definida por el número de dependencias ($L = 7$), el número de trenes ($T = 6$: $T_D = 3$, $T_U = 3$) y una de las nueve combinaciones de frecuencia F de la Tabla 7.1. El CSP generado para las 6 instancias fue: $\langle 86, 3600, 413, 2\rangle$.

Los resultados de la Tabla 7.3 muestran que en las instancias de F1 a F3, AC3 no pudo detectar que el problema era inconsistente, mientras que: 2-C3OPL y 2-C3 (ambas técnicas de 2-consistencia) detectaron la inconsistencia rápidamente. Esto es debido al hecho de que AC3 analiza cada restricción individualmente mientras que 2-C3 y 2-C3OPL analizan el conjunto de restricciones. De esta forma, 2-C3OPL y 2-C3 fueron más eficientes que AC3 porque detectaron tempranamente las inconsistencias y en el proceso realizaron pocos chequeos de restricciones y podas.

La Tabla 7.3 también muestra que el número de chequeos de restricciones en 2-C3OPL fue menor que en AC3 y 2-C3 en instancias consistentes (de F4 a F9). 2-C3OPL necesitó realizar menos chequeos de restricciones que 2-C3 y AC3, porque 2-C3OPL almacena los soportes encontrados en la matriz *Last*. Los chequeos de restricciones evitados por 2-C3OPL mejoran su tiempo de cómputo en un 70 % en relación con 2-C3 para instancias consistentes. Sin embargo, las estructuras adicionales requeridas por 2-C3OPL requieren tiempo, mientras que AC3 ahorra este tiempo.

Tabla 7.3: Número de podas, tiempo de cómputo (segundos) y número de chequeos de restricciones utilizando AC3, 2-C3 y 2-C3OPL en etapa de pre-proceso para el problema de asignación de horarios ferroviarios $\langle 86, 3600, 413, 2\rangle$, usando la combinación de trenes $T = T3$, el número de localidades $L = 7$ e incrementando la frecuencia F.

Frecuencia	**Arco-consistencia**			**2-consistencia**						Observación
	AC3			**2-C3**			**2-C3OPL**			
	podas	*tiempo*	*chequeos*	*podas*	*tiempo*	*chequeos*	*podas*	*tiempo*	*chequeos*	
F1	65300	31.6	2.3×10^9	I 3600	4.7	3.8×10^7	I 3600	4.8	3.8×10^7	Inconsistencia no es detectada por AC3
F2	65300	31.8	2.3×10^9	I 3600	4.7	3.8×10^7	I 3600	4.8	3.8×10^7	Inconsistencia no es detectada por AC3
F3	65300	31.6	2.3×10^9	I 3600	4.7	3.8×10^7	I 3600	4.8	3.8×10^7	Inconsistencia no es detectada por AC3
F4	70300	30.1	2.2×10^9	70300	289.3	2.6×10^9	70300	77.9	6.8×10^8	
F5	70300	30.3	2.2×10^9	70300	289.6	2.6×10^9	70300	77.8	6.8×10^8	
F6	70300	30.0	2.2×10^9	70300	289.4	2.6×10^9	70300	77.7	6.8×10^8	
F7	75300	28.7	2.1×10^9	75300	263.0	2.4×10^9	75300	76.1	6.8×10^8	
F8	75300	28.7	2.1×10^9	75300	263.1	2.4×10^9	75300	77.1	6.8×10^8	
F9	75300	28.7	2.1×10^9	75300	262.2	2.4×10^9	75300	76.2	6.8×10^8	

La Tabla 7.4 muestra el número de podas, tiempo de cómputo y número de chequeos de restricciones en el problema de asignación de horarios ferroviarios en etapa de pre-proceso, para nueve instancias diferentes en el recorrido *Zaragoza - Casetas*. Cada instancia fue definida por: $L = 7$, $T = 12$: $T_D = 6$, $T_U = 6$ y una de las nueve combinaciones de frecuencia F de la Tabla 7.1. La tupla de CSP generada fue: $\langle 170, 3600, 1376, 2 \rangle$.

A pesar del aumento de variables y restricciones, los resultados de la Tabla 7.4 son similares a los obtenidos en la Tabla 7.3 y nuevamente las técnicas de 2-consistencia propuestas son capaces de detectar las inconsistencias rápidamente.

Tabla 7.4: Número de podas, tiempo de cómputo (segundos) y número de chequeos de restricciones utilizando AC3, 2-C3 y 2-C3OPL en etapa de pre-proceso para el problema de asignación de horarios ferroviarios $\langle 170, 3600, 1376, 2\rangle$, usando la combinación de trenes $T = T8$, el número de localidades $L = 7$ e incrementando la frecuencia F.

Frecuencia	**Arco-consistencia**			**2-consistencia**						Observaciones
	AC3			**2-C3**			**2-C3OPL**			
	podas	*tiempo*	*chequeos*	*podas*	*tiempo*	*chequeos*	*podas*	*tiempo*	*chequeos*	
F1	161490	62.9	4.5×10^9	I 3600	4.8	3.8×10^7	I 3600	5.1	3.8×10^7	Inconsistencia no es detectada por AC3
F2	161490	63.0	4.5×10^9	I 3600	4.8	3.8×10^7	I 3600	5.1	3.8×10^7	Inconsistencia no es detectada por AC3
F3	161490	62.8	4.5×10^9	I 3600	4.9	3.8×10^7	I 3600	5.1	3.8×10^7	Inconsistencia no es detectada por AC3
F4	185990	57.3	4.1×10^9	185990	512.3	4.6×10^9	185990	151.0	1.3×10^9	
F5	185990	57.1	4.1×10^9	185990	512.5	4.6×10^9	185990	151.0	1.3×10^9	
F6	185990	57.3	4.1×10^9	185990	512.5	4.6×10^9	185990	151.0	1.3×10^9	
F7	210490	52.5	3.8×10^9	210490	462.6	4.2×10^9	210490	146.3	1.3×10^9	
F8	210490	52.7	3.8×10^9	210490	462.7	4.2×10^9	210490	146.3	1.3×10^9	
F9	210490	52.5	3.8×10^9	210490	462.2	4.2×10^9	210490	146.4	1.3×10^9	

La Tabla 7.5 muestra el número de podas, tiempo de cómputo y número de chequeos de restricciones en el problema de asignación de horarios ferroviarios en etapa de pre-proceso, para el recorrido *Zaragoza - Calatayud* en nueve instancias diferentes. Cada instancia estuvo definida por: $L = 5$, $T = 6$: $T_D = 3, T_U = 3$ y una de las combinaciones de frecuencia F de la Tabla 7.1. En estos problemas la talla de dominio se incrementó a 5000 segundos. La tupla de CSP generada fue: $\langle 62, 5000, 299, 2\rangle$.

Con estas instancias, la mejora en el número de chequeos de restricciones fue de un orden de magnitud con los algoritmos de 2-consistencia en relación a los algoritmos de arco-consistencia para los problemas consistentes, aunque el tiempo de cómputo fue peor.

Tabla 7.5: Número de podas, tiempo de cómputo (segundos) y número de chequeos de restricciones utilizando AC3, 2-C3 y 2-C3OPL en etapa de pre-proceso para el problema de asignación de horarios ferroviarios $\langle 62, 5000, 299, 2\rangle$, utilizando la combinación de trenes $T = T3$, el número de localidades $L = 5$ e incrementando la frecuencia F.

Frecuencia	**Arco-consistencia**			**2-consistencia**						Observación
	AC3			**2-C3**			**2-C3OPL**			
	podas	*tiempo*	*chequeos*	*podas*	*tiempo*	*chequeos*	*podas*	*tiempo*	*chequeos*	
F1	36356	39.3	2.8×10^9	I 5000	9.1	7.4×10^7	I 5000	9.2	7.4×10^7	Inconsistencia no es detectada por AC3
F2	36356	39.3	2.8×10^9	I 5000	9.1	7.4×10^7	I 5000	9.2	7.4×10^7	Inconsistencia no es detectada por AC3
F3	36356	39.3	2.8×10^9	I 5000	9.2	7.4×10^7	I 5000	9.3	7.4×10^7	Inconsistencia no es detectada por AC3
F4	40156	37.3	2.7×10^9	40156	380.0	3.3×10^9	40156	110.7	9.9×10^8	
F5	40156	37.3	2.7×10^9	40156	379.2	3.3×10^9	40156	111.0	9.9×10^8	
F6	40156	37.4	2.7×10^9	40156	379.8	3.3×10^9	40156	110.9	9.9×10^8	
F7	43956	36.0	2.6×10^9	43956	349.2	3.1×10^9	43956	110.2	9.8×10^8	
F8	43956	35.8	2.6×10^9	43956	349.5	3.1×10^9	43956	110.4	9.8×10^8	
F9	43956	35.8	2.6×10^9	43956	349.7	3.1×10^9	43956	110.4	9.8×10^8	

La Tabla 7.6 muestra el número de podas, tiempo de cómputo y número de chequeos de restricciones en el problema de asignación de horarios ferroviarios en etapa de pre-proceso, para el recorrido *Zaragoza-Casetas*, donde $L = 7$, $F = F7$ (ver Tabla 7.1) y el número de trenes fue incrementado de 6 a 12 (ver Tabla 7.2). Debido a que el parámetro T es incrementado, tanto el número de variables n, como el número de restricciones m, se incrementan en cada CSP resultante. La talla máxima de dominio d fue fijada en 3600.

Los resultados muestran que las técnicas de 2-consistencia propuestas realizan un 2 % más de poda que las de arco-consistencia (AC3). En el ratio de chequeos/podas, 2-C3OPL fue más eficiente que: AC3 y 2-C3. Por ejemplo, en el problema $< 86, 3600, 413, 2 >$, 2-C3OPL realizó 10364 chequeos por poda, mientras que 2-C3 y AC3 realizaron 41068 y 34774 chequeos por poda, respectivamente.

Tabla 7.6: Número de podas, tiempo de cómputo (seg.) y número de chequeos de restricciones utilizando AC3, 2-C3 y 2-C3OPL en etapa de pre-proceso para el problema de asignación de horarios ferroviarios, fijando el número de localidades $L = 7$, la frecuencia $F = F7$ y talla de dominio $d = 3600$ e incrementando el número de trenes T.

tupla CSP	**Arco-consistencia**			**2-consistencia**						Observaciones
	AC3			**2-C3**			**2-C3OPL**			
	podas	*tiempo*	*chequeos*	*podas*	*tiempo*	*chequeos*	*podas*	*tiempo*	*chequeos*	
$\langle 86, 3600, 413, 2\rangle$	65300	31.6	2.3×10^9	66340	311.3	2.7×10^9	66340	78.4	6.7×10^8	menor poda con AC3
$\langle 114, 3600, 720, 2\rangle$	93798	42.4	3.0×10^9	95838	415.7	3.7×10^9	95838	105.9	9.3×10^8	menor poda con AC3
$\langle 142, 3600, 1048, 2\rangle$	125868	52.8	3.8×10^9	129208	487.2	4.4×10^9	129208	130.4	1.1×10^9	menor poda con AC3
$\langle 170, 3600, 1376, 2\rangle$	161490	62.7	4.5×10^9	166450	560.3	5.0×10^9	166450	154.3	1.3×10^9	menor poda con AC3

7.2.2. Evaluación de técnicas de búsqueda

Para la búsqueda de una solución al problema de asignación de horarios ferroviarios, se utilizaron los siguientes algoritmos de búsqueda genéricos: BT, FC, RFLA[2], y se compararon los resultados con los algoritmos propuestos: BLS y SchTrains. Todos los algoritmos realizaron consistencia en etapa de pre-proceso (previo a realizar la búsqueda). Así, RFLA y FC se ejecutaron con AC3 mientras que SchTrains, BLS y BT se ejecutaron utilizando 2-C3OPL.

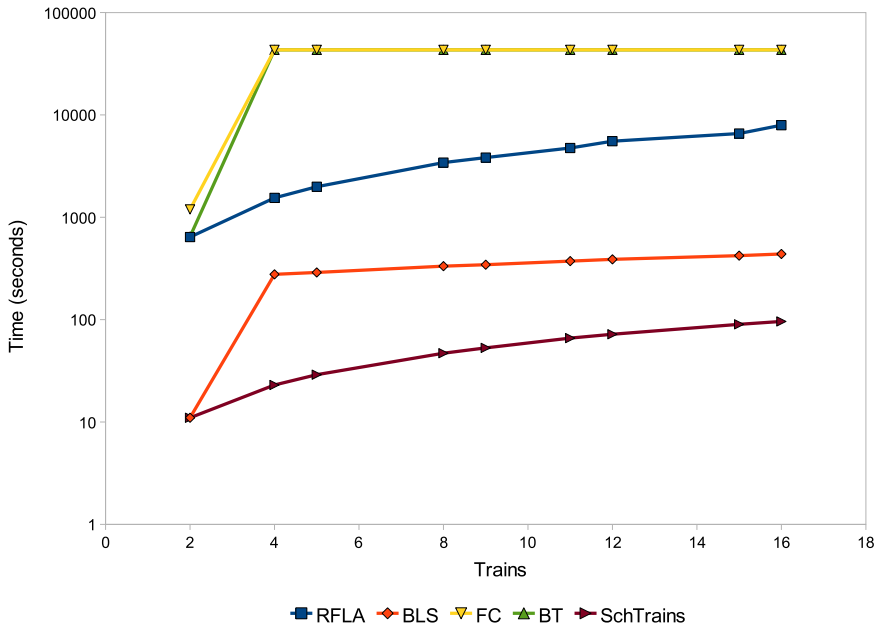


Figura 7.5: Tiempo de cómputo (seg.) empleado por los algoritmos de búsqueda BT, FC, RFLA, BLS y SchTrains para el problema de asignación de horarios ferroviarios en el recorrido Zaragoza-Casetas, fijando $L = 7$ y $F = F2$ e incrementando T.

La Figura 7.5 muestra los resultados de la búsqueda en el tramo Zaragoza-Casetas utilizando los algoritmos BT, FC, RFLA, BLS y SchTrains. El número de trenes T se incrementó de 2 a 16 utilizando las combinaciones de la Tabla 7.2. Para

[2]La búsqueda con RFLA se realizó utilizando el resolutor CON'FLEX. Dicho resolutor está disponible en http://www.inra.fr/internet/Departements/MIA/T/conflex

todas las instancias el número de dependencias se fijó a 7; la talla del dominio se fijó a $d = 2500$; Las frecuencias de salida y llegada se fijaron a $F_D = 100$ y $F_U = 120$, respectivamente; el retraso se fijó a 5 y el tiempo máximo de finalización se fijó en 43200 segundos. Cada problema genera diferentes instancias de CSP, así por ejemplo, con la combinación de trenes $T1$, el CSP generado es $\langle 58, 2500, 77, 2\rangle$ y con la combinación de trenes $T10$, el CSP generado es $\langle 226, 2500, 434, 2\rangle$. Como puede observarse en la gráfica, en la búsqueda de una solución el algoritmo SchTrains tuvo mejor desempeño en todas las instancias que los algoritmos RFLA, FC, BT y BLS. Esto es debido a que el proceso de búsqueda es guiado con el conocimiento que se dispone del dominio, y SchTrains inicia con instanciaciones válidas de la poda proporcionada por el proceso de consistencia (2-C3OPL) realizado en la etapa de pre-proceso. Así mismo, el algoritmo BLS tuvo mejor desempeño, en todas las instancias, que los algoritmos RFLA, FC y BT. Ello se debe a que BLS utiliza la información almacenada en el último soporte encontrado para guiar la búsqueda y no requiere rehacer el proceso de consistencia que si es hecho por FC y RFLA (en dominios grandes, rehacer la consistencia varias veces es muy costoso en lo que se refiere a tiempo de cómputo). También, en instancias superiores a 4 trenes, los algoritmos FC y BT alcanzaron el tiempo máximo de finalización sin generar la solución.

Finalmente, la Figura 7.6 muestra los resultados de búsqueda en el recorrido Zaragoza-Calatayud, utilizando los algoritmos BT, FC, RFLA, BLS y SchTrains. Para estos experimentos el número de trenes se fijo a 6; las frecuencias de salida y llegada se fijaron a $F_D = 100$ y $F_U = 120$, respectivamente; el retraso se fijó a 5 y el tiempo máximo de finalización se fijó en 86400 segundos. Se utilizaron dos tallas de dominio: $d = 2500$ para instancias de 7 a 15 dependencias, y $d = 5000$ para instancias de 20 a 25 dependencias. El cambio de dominios se debe a que en problemas con 20 dependencias y 6 trenes, en un dominio de 2500 no hay solución[3] y en cambio, aumentando la talla de dominio a $d = 5000$ se puede encontrar una solución. Las distintas instancias obtenidas están en el rango de $\langle 86, 2500, 126, 2\rangle$ a $\langle 302, 5000, 342, 2\rangle$. Nuevamente, en la búsqueda de una solución, el algoritmo SchTrains tuvo mejor desempeño que los algoritmos RFLA, FC, BT y BLS en todas las

[3]Por ejemplo: el tiempo de cómputo para problemas sin solución (seg.) con 25 dependencias y talla de dominio de 2500 fueron: RFLA=348; FC=658; BT=5; BLS=5 y SchTrains=5.

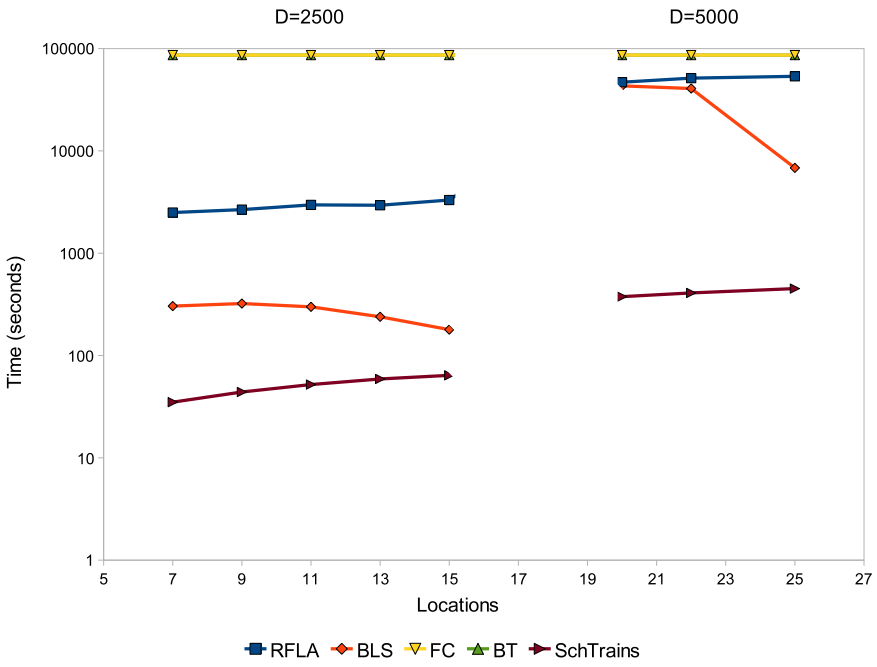


Figura 7.6: Tiempo de cómputo (seg.) utilizando en la búsqueda BT, FC, RFLA, BLS y SchTrains para el problema de asignación de horarios ferroviarios en el trayecto Zaragoza-Calatayud, fijando $T = T3$ y $F = F2$ e incrementando L.

instancias. El algoritmo BLS tuvo mejor desempeño que los algoritmos RFLA, FC y BT en instancias de 7 a 15 dependencias y con 25 dependencias, pero en instancias con 20 y 22 dependencias BLS no fue capaz de encontrar solución (por ser un algoritmo heurístico). Así mismo, puede observarse que en las instancias de $L = 9$ a $L = 15$, el tiempo de cómputo requerido por BLS para encontrar una solución decrece, lo cual es debido a que cada tren pasa por más dependencias y extiende su tiempo de recorrido. Esto es utilizado por BLS en asignar valores a cada variable consistentes con su variable padre, permitiendo explorar más rápidamente el espacio de búsqueda y encontrar la solución.

7.3. Conclusiones

El problema de asignación de horarios ferroviarios es un problema real que puede ser modelado como un CSP. Esta formulación genera problemas grandes y de amplios dominios en los que las técnicas de consistencia son importantes para podar el espacio de búsqueda y para mejorar el proceso de búsqueda.

En este capítulo hemos presentado el desempeño de las técnicas de consistencia propuestas (2-C3 y 2-C3OPL) combinadas con las técnicas de búsqueda propuestas que trabajan en CSP binarios no-normalizados (algoritmo Loop-Back Last Search, BLS y el algoritmo SchTrains) en un problema real.

Los algoritmos 2-C3OPL y 2-C3 se aplicaron al problema de asignación de horarios ferroviarios debido a que en su versión no-normalizada, el problema se beneficia con una poda mayor como la que realiza la 2-consistencia. La evaluación mostró que 2-C3OPL tuvo mejor desempeño que AC3 y 2-C3 en este tipo de problemas. Adicionalmente, AC3 fue incapaz de detectar inconsistencias, mientras que 2-C3OPL y 2-C3 si pudieron hacerlo eficientemente. Los operadores ferroviarios consideran interesante la detección de inconsistencias en horarios preliminares, lo cual hace útil las técnicas de 2-consistencia propuestas. Así mismo la evaluación muestra que el tiempo de cómputo fue reducido sobre un 75 % y el número de chequeos de restricciones se redujo en 50 % en relación a 2-C3 en instancias consistentes.

En la búsqueda el algoritmo BLS tuvo mejor desempeño que los algoritmos RFLA, FC y BT, en varias instancias del problema de asignación de horarios ferroviarios, donde el tiempo de cómputo fue reducido en un 80 % con respecto a RFLA y fue reducido en un 99 % con respecto a FC y BT en instancias con solución. No obstante, el algoritmo BLS no fue capaz de encontrar una solución en un 12 % de los problemas de trenes evaluados. El algoritmo SchTrains tuvo mejor desempeño que los algoritmos RFLA, FC, BT y BLS en todas las instancias del problema de asignación de horarios ferroviarios, la cual estuvo sobre un orden de magnitud.

Capítulo 8

Modelización y resolución vía CSP a problemas de Planificación y Scheduling

8.1. Introducción

Un problema de planificación y scheduling $P_{P\&S}$ es un problema de búsqueda en un espacio complejo que posee restricciones y hay que optimizar una función objetivo, lo que implica un problema combinatorio en cuanto al número de alternativas posibles que conduzcan hacia los objetivos.

Definición 8.1.1. Un *problema de Planificación y Scheduling* (P&S) se puede definir mediante la tupla $P_{P\&S} = \langle S_I, A, S_F, R, F_O \rangle$, donde S_I representa el conjunto inicial de hechos que forman la situación inicial del problema, A representa el conjunto de acciones definidas en el dominio del problema, S_F representa el conjunto final de hechos que forman la situación final del problema; R representa el conjunto de recursos disponibles para la ejecución de un plan y F_O es la función objetivo a optimizar.

Intuitivamente, la resolución del problema de $P_{P\&S}$ consiste en alcanzar el estado final S_F partiendo de la situación inicial S_I, aplicando las acciones A que utilizan los recursos R y optimizando el valor de F_O.

Definición 8.1.2. Un *operador de planificación* o está definido por la terna $o = (nombre(o), precondiciones(o), efectos(o))$, donde *nombre* contiene el nombre del operador y la lista de parámetros $r_1...r_n$ de los que depende el operador o, las *precondiciones* son el conjunto de parámetros que deben ser ciertos antes de que se aplique el operador o y los $efectos$ son el conjunto de parámetros que se añaden o eliminan al aplicar el operador o.

Ejemplo. El dominio rocket (cohete espacial) tiene tres operadores de planificación: *Load* (cargar), *Unload* (descargar), y *Move* (mover). Veamos el operador *Load*: una pieza de carga ?c puede ser cargada (Load) en el rocket ?r si el rocket y y la pieza de carga están en el mismo lugar ?p. Podemos escribir el operador *Load* de la siguiente forma:

(define (operator *Load*)
:parameters ((rocket ?r) (lugar ?p) (carga ?c))
:precondition (:and (at ?r ?p) (at ?c ?p))
:effect (:and (:not (at ?c ?p)) (in ?c ?r)))

Los símbolos ?r, ?p y ?c indican que los parámetros tipo *rocket*, *lugar* y *carga* no han sido instanciados. Un problema típico puede tener uno o más rockets y varias piezas de carga en el lugar de inicio con el objetivo de mover la carga a diferentes lugares de destino.

Definición 8.1.3. Una *acción* a es un operador de planificación o instanciado, es decir, la lista de parámetros del operador contiene valores constantes de la clase apropiada. La acción ficticia *Start* no tiene precondiciones y sólo contiene efectos, y la acción ficticia *End* sólo contiene precondiciones y no contiene efectos.

Una *acción es aplicable* en el momento que sus precondiciones son satisfechas.

Ejemplo. Para el dominio rocket mostrado anteriormente, si el problema tiene dos rockets $r1$ y $r2$, tres lugares $l1, l2$ y $l3$, dos piezas de carga $c1$ y $c2$, donde al

inicio el rocket $r1$ y la pieza de carga $c1$ están ubicados en $l2$ y el rocket $r2$ y la pieza de carga $c2$ están ubicadas en el lugar $l3$, el objetivo es colocar todas las piezas de carga en $l1$. Con estos datos al instanciar el operador *Load* se crean 12 acciones, una de las cuales sería:

(:action *Load-r1-l2-c1*
:parameters()
:precondition (and $(at - r1 - l2)$ $(at - c1 - l1)$)
:effect (and (not $(at - c1 - l1)$) $(in - c1 - r1)$))

Al inicio, la acción *Load-r1-l2-c1* es una acción aplicable, ya que tiene satisfechas sus precondiciones.

En la resolución de un problema real de $P_{P\&S}$, podemos distinguir tres etapas:

1. Etapa de Modelado del problema, utilizando un lenguaje que permita representar de forma fácil y completa el problema a resolver.

2. Etapa de Planificación, donde se obtiene un plan de acciones sintáctica y semánticamente correctos.

3. Etapa de Scheduling: donde se fijan las acciones en el tiempo, se comprueba la viabilidad del plan en relación a los recursos disponibles y se optimiza la función objetivo.

En este capítulo nos centraremos en la Etapa 1 del proceso de resolución de un problema de $P_{P\&S}$: Modelado del problema. Así mismo, para ilustrar nuestra propuesta de lenguaje de modelado lo haremos desde la perspectiva del planificador, específicamente de los planificadores POCL (de las siglas del inglés: Partial Order Causal Link) utilizando la filosofía de los CSP. Cabe destacar que la propuesta de lenguaje de modelado que realizamos es una extensión CPT [121], formulada por Vidal y Geffner.

8.2. Planificación POCL

A comienzos de la década de los 90, gran parte de la investigación en planificación independiente del dominio se centró en los planificadores POCL (Partial Order Causal Link), entre los que destacan SNLP desarrollado por McAllester y Rosenlitt y UCPOP, desarrollado por Penberthy y Weld [109].

Las técnicas empleadas por estos planificadores se basan en una búsqueda regresiva sobre un *espacio de planes*, donde el planificador se mueve y selecciona, basándose en un conjunto de criterios, cual es el mejor plan a expandir. Un espacio de planes es un grafo dirigido implícito, cuyos vértices son planes parciales y sus aristas corresponden a operaciones de refinamiento. La forma como se realiza una operación de refinamiento es mostrada en la Figura 8.1.

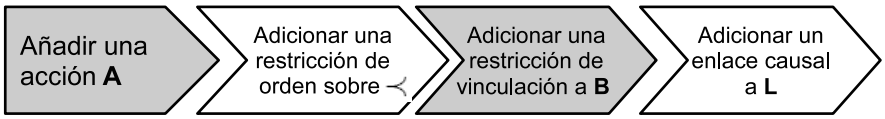


Figura 8.1: Operación de refinamiento de Planificación POCL.

Definición 8.2.1. Un *enlace causal* entre dos acciones a_i y a_j es un plan escrito como $a_i \xrightarrow{p} a_j$. Se lee como a_i alcanza a a_j a través de p, donde p es un efecto producido por a_i y una precondición para a_j .

Definición 8.2.2. Una acción a_k entrará en conflicto con $a_i \xrightarrow{p} a_j$ si a_k contiene $\neg p$, y si a_k pudiera traer a_i antes que a_j.

Definición 8.2.3. Una *precondición p es abierta* (o está no resuelta) si no es alcanzada por ninguna acción en el plan.

El proceso de planificación POCL se desarrolla de la siguiente forma:

1. Comienza con un plan inicial que sólo contiene las acciones *Inicial* (*Start*) y *Final* (*End*).

2. En cada iteración se selecciona una acción a_j que tenga precondiciones no resueltas.

3. Se escoge una de las precondiciones p de la acción a_j seleccionada, como el siguiente sub-objetivo a resolver. Para resolverlo se puede añadir una nueva acción o utilizar una acción existente en el plan a la que llamaremos a_i y que produzca como efecto la precondición p. Esto hace que se produzca un enlace causal $a_i \xrightarrow{p} a_j$

4. Si debido a este proceso, llega una acción a_k ordenada entre las acciones a_i y a_j ($a_i \prec a_k \prec a_j$) que elimina el efecto p de la acción a_i, aparece una amenaza, la cual puede ser resuelta añadiendo una restricción de orden. Las restricciones de orden son:

   - por democión ordenamos que la acción conflictiva a_k este posterior a la acción a_j, i.e., $a_j \prec a_k$
   - por promoción: se coloca la acción conflictiva a_k antes que la acción a_i, es decir, $a_k \prec a_i$.

5. Cuando no es posible realizar la promoción o democión de las restricciones, pues ello supone una violación de los órdenes ya existentes, se descarta el plan en cuestión y se regresa al punto (2).

6. El proceso finaliza cuando se consigue un plan en el cual todos sus sub-objetivos han sido resueltos y no existen conflictos entre las acciones del mismo.

Pese a que en un principio estos planificadores sólo podían trabajar con muy pocas acciones, gracias a la aplicación de heurísticas, como las de análisis de alcanzabilidad y estimación de distancias hasta el objetivo, se mejora la eficiencia de los planificadores POCL haciéndolos competitivos con los planificadores basados en estados.

El paradigma de POCL se puede extender para incluir y manejar el tiempo en la planificación [53], ya que POCL es independiente de la asunción de que las acciones deban ser instantáneas o tengan la misma duración; el orden parcial y los mecanismos de preservación de sub-objetivos le permiten a POCL definir acciones de duración arbitraria, siempre que estén bien definidas, las condiciones bajo las

cuales interfieren las acciones y el uso del mínimo-compromiso (least-commiment) le permite ofrecer un alto grado de flexibilidad en comparación a otros sistemas de planificación temporal.

La mayoría de los planificadores temporales POCL comparten un marco representativo común: la representación de intervalos para acciones y proposiciones, la cual permite por ejemplo: especificar que una condición es necesitada solo durante una parte de la ejecución de la acción; o, que una acción pueda ser ejecutada mientras se ejecuta otra acción; o, que una acción a deba ser efectuada t unidades de tiempo después de la finalización de una acción b ; lo cual ofrece una representación de modelado más rica que la que ofrece PDDL (de las siglas del inglés: Planning Domain Definition Language).

Esta expresividad fue explotada por los planificadores Zeno [92] e IxTeT [67]. El primero, desarrollado por Penberty y Weld, soporta características métricas como: razonamiento temporal, tiempos máximos en la resolución de objetivos y eventos exógenos. El segundo, desarrollado por Ghallab y Laurelle, permite precondiciones temporales y el acceso a puntos de tiempo en el intervalo de duración de una acción. Pero dicha expresividad hace que los algoritmos que detectan y resuelven amenazas se tornen más complejos, por la representación y manejo de restricciones temporales entre acciones.

8.3. Métodos generales de solución en la planificación temporal POCL

Se distinguen dos métodos generales de solución en la planificación temporal POCL: la extensión del algoritmo POCL clásico, para que trabaje con restricciones temporales y proposiciones; y los que combinan técnicas de CSP con planificación POCL.

1. **Extensión del algoritmo clásico POCL:** la idea es incorporar el razonamiento temporal al planificador POCL, separando la parte temporal en una red temporal TMM (Time Map Manager) de la parte procedural de la construcción del plan. Su implementación no es trivial, ya que la resolución de

conflictos es más compleja por el gran número de restricciones. FORBIN, O-Plan y TRIPTIC son planificadores POCL que trabajan de este modo.

2. **Combinación de las técnicas de CSP y planificación POCL:** dependiendo del papel que jueguen el CSP y el planificador, así como de la forma en que se realice la modelización y posterior resolución, podemos distinguir dos enfoques:

   a) **Enfoque de cooperación entre el planificador POCL y el resolutor de CSP**, para resolver juntos el problema. Las técnicas de satisfacción de restricciones son añadidas como un adjunto al proceso de planificación, pero dicho proceso no es formulado como un CSP. En estas aproximaciones cada plan generado por el planificador POCL es representado como una STN (de las siglas en ingles: Simple Temporal Network). La estructura lógica del plan es generada por el algoritmo de POCL y el resolutor CSP es el responsable de verificar la consistencia de las restricciones temporales, los recursos y la métrica.

   b) **Enfoque de formulación de programación de restricciones para planificación POCL:** la idea es modelar el problema de planificación POCL como un CSP, con variables, dominios y restricciones, las cuales incluyen mecanismos de razonamiento para manejar enlaces causales, relaciones de exclusión mutua, relaciones de orden y amenazas, para ser resuelto aplicando técnicas de CSP. Esta formulación permite enriquecer al modelo POCL, al permitir que se codifiquen restricciones complejas sobre recursos numéricos, proposiciones y acciones del plan. Los pasos básicos en esta formulación son: (i) formular las variables que representan a las acciones, proposiciones y fluents (recursos numéricos) ; (ii) formular las restricciones; (iii) formular los mecanismos de bifurcación (iv) formular los mecanismos de poda que permitan reducir el espacio de búsqueda. El resolutor CPT desarrollado por Vidal y Geffner [121], y la aplicación que desarrollamos en esta tésis, se basan en este enfoque.

8.4. Expresividad del lenguaje de modelado

La modelado con CSPs facilita la especificación de restricciones complejas. Adicional a las características propuestas por PDDL, de proporcionar una semántica completa para planes temporales que incluye representación explícita del tiempo, la duración de las acciones y el consumo de recursos continuos, predicados derivados y ventanas temporales (timed inicial literals), restricciones blandas y duras, el lenguaje de modelado que proponemos permite representar nuevas características presentes en problemas reales, como las se listan a continuación:

1. **Modelo más elaborado de acciones**, que va más allá del modelo conservativo de acciones utilizado en [121] y el no-conservativo utilizado en PDDL 2.1 [81], PDDL 2.2 [45] y PDDL 3.0 [64], que permite modelar restricciones del mundo real, donde:

   - **Duración:** puede ser un valor fijo o variar en un intervalo de valores conocido (ver Fig. 8.2, (a)).
   - **Condiciones:** pueden ser requeridas en cualquier momento, dentro o fuera del intervalo de la acción[1] (ver Fig. 8.2, (b)).
   - **Efectos:** al igual que las condiciones, puedan ser generados en cualquier momento, dentro o fuera del intervalo de la acción (ver Fig. 8.2, (c)).

   Este modelo más elaborado de acciones permite modelar condiciones y efectos que no están inmersas dentro de la duración de la acción, lo cual facilita el modelado de situaciones reales sin necesidad de añadir nuevas acciones.

2. **Restricciones complejas**: adicional a las restricciones que se pueden modelar en PDDL [17, 18 y 19], el modelo propuesto permite:

   - **Restricciones de precedencia entre acciones**: adicional a las restricciones cualitativas implícitas de los enlaces causales, se pueden modelar explícitamente restricciones de orden entre acciones.
   - **Restricciones temporales entre proposiciones, acciones y proposiciones - acciones:** similar a las restricciones de precedencia, el modelo

[1] Utilizamos el término Condiciones en lugar de precondiciones debido a que estas últimas sólo son pueden ser requeridas antes de iniciarse la acción

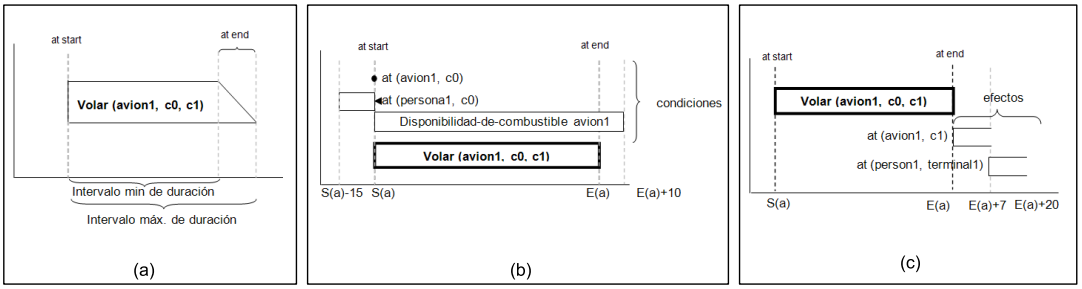


Figura 8.2: Representación del modelo extendido para la acción *Volar*: (a) Duración; (b) Condiciones y (c) Efectos.

considera restricciones temporales cuantitativas entre el tiempo de generación de las proposiciones, el tiempo de inicio/finalización de las acciones y cualquier combinación entre ellas.

- **Tiempos máximos de finalización (deadlines) para acciones y objetivos:** que pueden restringir la estructura del plan y que permiten alcanzar objetivos intermedios que no tienen que persistir hasta que finalice el plan y modelar objetivos de alto nivel a ser satisfechos en cualquier tiempo durante la ejecución del plan.

- **Ventanas temporales sobre proposiciones/acciones:** Restricciones externas que pueden incluir limitaciones sobre la disponibilidad de algunas proposiciones e intervalos de tiempo cuando las acciones deben ser ejecutadas.

3. **Habilidades numéricas para expresar el uso de los recursos continuos:** (fluents) con el modelado de ecuaciones e inecuaciones tanto en condiciones como en efectos.

8.5. Formulación CSP en Planificación POCL para problemas $P_{P\&S}$

La formulación que se presenta está basada en CPT [121] e incluye el modelo de acciones y restricciones complejas propuesto en la sección anterior, por lo que en esta sección se realizará especial énfasis en su codificación.

La idea es codificar automáticamente un problema de $P_{P\&S}$ en un CSP con variables, dominios y restricciones. Así, las entradas serán: el *Dominio* de acciones con sus relaciones (enlaces causales restricciones de soporte, amenazas y relaciones de exclusión mutua -mutex-); el *Problema* que especifica el estado inicial y el estado objetivo. Al problema se le pueden adicionar: recursos numéricos, restricciones temporales, restricciones adicionales complejas, métrica de optimización multi-criterio, plan solución (realizado por el usuario u otra herramienta automatizada de planificación), para que dicha formulación sea resuelta por una herramienta resolutora de CSPs para encontrar un plan solución (ver Fig. 8.3).

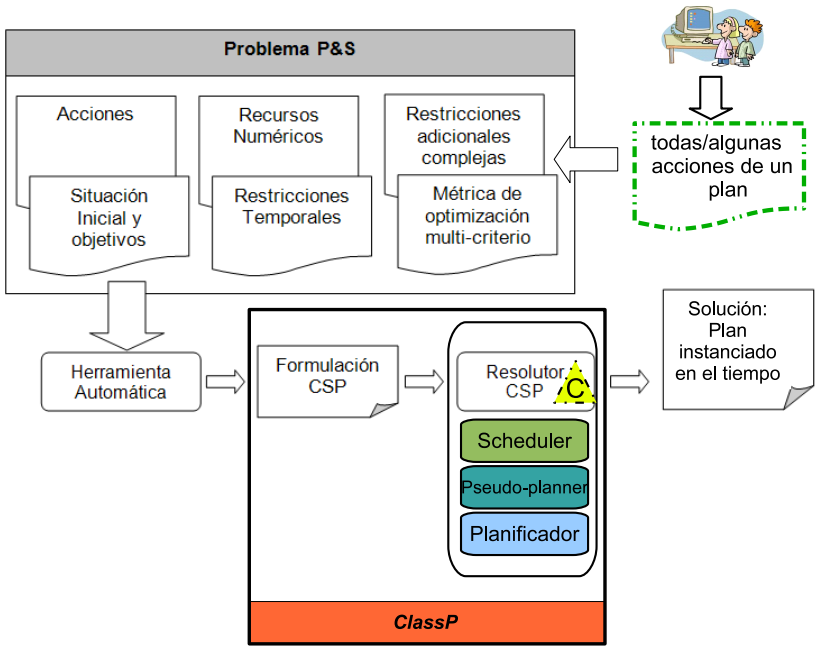


Figura 8.3: Esquema general del proceso de resolución de un problema de $P_{P\&S}$ en una formulación de programación de restricciones y la obtención del plan.

El usuario puede decidir darle o no más trabajo al resolutor CSP: (a) indicando

en el problema $P_{P\&S}$ la presencia de un plan solución a optimizar, *modo scheduler*, para que el resolutor CSP sólo realice la asignación de puntos de tiempo y recursos; (b) indicando que una o más acciones formarán parte del plan, *modo pseudo-planner*, donde el resolutor CSP debe asignar el resto de las acciones y soportes para la consecución de un plan válido, o bien (c) otorgándole toda la libertad al resolutor CSP para que escoja las acciones y soportes válidos, *modo planificador o selector de acciones*, actuando de este modo en forma de planificador con capacidades de scheduling capaz de asignar tiempo y recursos en restricciones complejas. Así mismo, se le descargará trabajo al resolutor si el usuario aplica una de las técnicas de filtrado explicadas en los capítulos precedentes, lo cual puede realizarse adicionando la técnica de filtrado seleccionada al resolutor.

Para explicar la formulación propuesta, la hemos dividido en cuatro grupos:

1. Variables y Dominios.
2. Restricciones I: Generales.
3. Restricciones II: Situaciones de Bifurcación.
4. Heurísticas.

8.5.1. Variables y Dominios

Las variables son básicamente utilizadas para definir las acciones a y las condiciones requeridas. Denotaremos a las condiciones proposicionales p y a las condiciones numéricas Φ (fluents); los soportes sobre las condiciones de las acciones y el tiempo (modelado en $\mathbb{R}$) en el cual ocurren esas condiciones.

Se incluyen dos acciones ficticias *Start* y *End* (Inicio y Fin), donde la acción *Start* soporta todas las proposiciones y fluents que posee el estado inicial (i.e., son los efectos proporcionados por la acción *Start*), mientras que la acción *End* requiere como condiciones las proposiciones y fluents de los objetivos (goals) del problema a resolver.

Los dominios, son el conjunto de valores que puede tomar una variable. Los dominios pueden contener valores numéricos comprendidos en un intervalo entre $(-\infty, \infty)$ o también acciones.

Las variables, sus dominios iniciales y su descripción serán organizadas en tres subgrupos: *Grupo V1*, que comprende las variables necesarias para cualquier acción; *Grupo V2*, que comprende las variables necesarias para la parte proposicional de la acción y *Grupo V3*, que comprende las variables para la parte numérica de la acción.

Grupo V1: Variables necesarias para cualquier acción

- $S(a), E(a) \in [0, \infty)$ representan el tiempo de inicio y finalización de la acción a. Claramente, $S(Start) = E(Start)$ y $S(End) = E(End)$.
- $Dur(a) \in [Dur_{min}(a), Dur_{max}(a)]$ representa la duración de la acción a entre dos valores límites positivos. Evidentemente, $Dur(Start) = Dur(End) = 0$.
- $InPlan(a) \in [0, 1]$ codifica una variable binaria que indica la presencia de la acción a en el plan solución ($1 \Rightarrow$ la acción estará en el plan y $0 \Rightarrow$ la acción no está en el plan). Evidentemente, $InPlan(Start) = InPLan(End) = 1$.

El resto de las acciones de los grupos V2 y V3 son consideradas condicionalmente. En este sentido, las variables que son significativas son aquellas cuyo $InPlan(a) = 1$. Cabe destacar que CPT [121] considera la duración de una acción como un valor y no un intervalo de valores y que la acción $InPlan(a)$ no está presente en su codificación.

Grupo V2: Variables necesarias para la parte proposicional de la acción

Estas variables sólo son necesarias si la acción tiene condiciones y/o efectos proposicionales p.

- $Sup(p, a) \in \{b_i\}$ tal que b_i adiciona p. Representa un soporte potencial para la acción a, es decir, es el enlace causal $b_i \xrightarrow{p} a$. Inicialmente, incluye a todas las acciones posibles que soporten p y $Sup(p, a)$ será el que finalmente asigne el valor a la variable $Time(p, a)$.
- $Time(p, a) \in [0, \infty)$ representa el tiempo cuando sucede el enlace causal $Sup(p, a)$.
- $Persist(p, a) \in [0, \infty)$ representa la persistencia de p para la acción a.

- $Req_{start}(p,a)$, $Req_{end}(p,a) \in [0,\infty)$ representan el intervalo de tiempo en el cual la acción a requiere a p, con $Req_{start}(p,a) \leq Req_{end}(p,a)$, y donde usualmente $Req_{start}(p,a) = S(a)$ y $Req_{end}(p,a) = E(a)$. Esta formulación le permite al usuario representar un amplio tipo de condiciones, desde las condiciones puntuales $Req_{start}(p,a) = Req_{end}(p,a)$ hasta las condiciones que salen del intervalo de duración de la acción, como por ejemplo $Req_{start}(p,a) = S(a) - 1$ y $Req_{end}(p,a) = E(a) + 1$, donde p es requerido en el intervalo que transcurre desde una unidad de tiempo antes de iniciarse la acción a y hasta una unidad de tiempo posterior a la finalización de dicha acción a.

Cabe destacar que las variables $Req_{start}(p,a)$ y $Req_{end}(p,a)$ extienden el modelo conservativo de CPT [121] donde las precondiciones de la acción sólo se requieren al inicio de la acción y los efectos se generan al final de la misma.

Grupo V3: Variables necesarias para la parte numérica de la acción

Estas variables sólo son necesarias: si la acción tiene condiciones y/o efectos numéricos Φ(fluents) y para la optimización de la métrica.

- $Sup(\Phi,a) \in \{b_i\}|b_i$ soporte del fluent Φ para la acción a. Similar al $Sup(p,a)$, esta variable simbólica codifica el soporte de b_i para el fluent Φ.

- $Time(\Phi,a) \in [0,\infty)$ representa el tiempo cuando sucede el enlace causal $Sup(\Phi,a)$.

- $Req_{start}(\Phi,a)$, $Req_{end}(\Phi,a) \in (-\infty,\infty)$ representan el intervalo de tiempo en el cual la acción a debe satisfacer la condición numérica $Cond(\Phi,a)$.

- $V_{actual}(\Phi,a) \in (-\infty,\infty)$ representa el valor actual del fluent Φ al tiempo $Req_{start}(\Phi,a)$, si la acción a requiere una restricción de condición numérica $Cond(\Phi,a)$, en caso contrario, $V_{actual}(\Phi,a)$ representa el valor actual del fluent Φ en el tiempo cuando se inicia la acción $S(a)$.

- $V_{updated}(\Phi,a) \in (-\infty,\infty)$ representa el valor cuando la acción a actualiza el fluent Φ. Esta acción sólo es necesaria si la acción a modifica al fluent Φ.

Es importante destacar que la extensión numérica de la acción no es soportada por CPT [121] y que cuando se trabaja con fluents el término soporte no es el más apropiado, ya que no existe una acción en particular que soporte al fluent Φ y muchas acciones $\{a_i\}$ pueden actualizarlo. Así, la variable $Sup(\Phi, a)$ contendrá la última acción a_i que actualizó el fluent Φ antes de ejecutar la acción a, propagando así su valor para la acción a. Adicionalmente, cuando este valor se propaga, la variable $V_{actual}(\Phi, a)$ es necesaria en los siguientes casos:

- Si existe una condición numérica de Φ para la acción a, como por ejemplo $Cond(\Phi, a) = V_{actual}(\Phi, a) > 100$, entonces la variable $V_{actual}(\Phi, a)$ almacenará el valor Φ que debe satisfacer la condición numérica.

- Si existe un efecto numérico de Φ para la acción a, entonces $V_{actual}(\Phi, a)$ almacenará el valor inicial para calcular el valor actualizado de $V_{updated}(\Phi, a)$ con los efectos del incremento o decremento realizados por la acción a. Consecuentemente, el valor de $V_{updated}(\Phi, a)$ podrá basarse en una modificación de $V_{actual}(\Phi, a)$ o en la asignación de un valor absoluto.

8.5.2. Restricciones I: Generales

El modelo propuesto genera un gran número de restricciones, pero sólo se considerarán aquellas restricciones con acciones que pertenezcan al plan, $(\forall a : Inplan(a) = 1)$. Las restricciones pertenecientes al grupo Restricciones I, las hemos organizado en cuatro subgrupos:

Grupo RI.1: Restricciones necesarias para todas las acciones

- $S(a) + Dur(a) = E(a)$ indica el inicio y la finalización de la acción a.

- $E(Start) \leq S(a)$ indica que la acción a debe comenzar después de la acción ficticia $Start$.

- $E(a) \leq S(End)$ indica que la acción a debe finalizar antes de la acción ficticia End.

Grupo RI.2: Restricciones necesarias para la parte proposicional de las acciones

- $Time(p, a) \leq Req_{start}(p, a)$ fuerza que la condición p esté antes de que sea requerida por la acción a, estableciendo un enlace causal.
- Si $Sup(p, a) = b_i$, entonces $Time(p, a) =$\{tiempo cuando la acción b_i añade p\}, coloca el tiempo inicial para satisfacer a $Sup(p, a) = b_i$.
- Si $Sup(p, a) = b_i$, entonces $Persist(p, a) =$ persistencia de p proporcionada por la acción b_i.

Grupo RI.3: Restricciones necesarias para la parte numérica de las acciones

- $Time(\Phi, a) \leq Req_{start}(\Phi, a)$ fuerza a que el fluent Φ se satisfaga antes de que sea requerido por la acción a, estableciendo un enlace causal.
- Si $Sup(\Phi, a) = b_i$, entonces $Time(\Phi, a) =$ tiempo cuando la acción b_i que soporta a la acción a actualiza a Φ, coloca el tiempo para satisfacer $Sup(\Phi, a)$.
- Si $Sup(\Phi, a) = b_i$, entonces $V_{actual}(\Phi, a) = V_{updated}(\Phi, a)$ determina como $V_{updated}(\Phi, a)$ es propagado a lo largo del modelo.
- $Cond(\Phi, a) = V_{actual}(\Phi, a)$ op Exp, donde $op = \{<, \leq, =, \geq, >, \neq\}$ y Exp es cualquier combinación de variables y/o valores que son evaluados en $\mathbb{R}$. Representa la condición numérica Φ que debe cumplirse para la acción a en el intervalo que sea requerida.

Grupo RI.4: Restricciones adicionales complejas

- $Req_{end}(p, a) \leq Persist(p, a)$ representa las restricciones de persistencia, estableciendo que el límite superior del intervalo de un requerimiento de una condición nunca exceda el valor de la persistencia de dicha condición.
- $min(tw(p)) \leq Req_{start}(p, a) \leq Req_{end}(p, a) \leq max(tw(p))$, representa una ventana temporal para la proposición p (o también puede ser un fluent, si se

sustituye p por Φ) y $min(tw(a)) \leq S(a) \leq E(a) \leq max(tw(a))$, representa una ventana temporal para una acción a.

- Var_i op $Var_j + x$ donde $op = \{<, \leq, =, \geq, >, \neq\}$ y $x \in \mathbb{R}$, representa una restricción binaria.

- $Constraint(Var_i, Var_j, ..., Var_m, Var_n)$ representa una restricción n-aria personalizada que puede codificar restricciones más complejas sobre variables del modelo, dependiendo de un problema particular.

Este grupo de restricciones complejas son raramente manejadas en la planificación tradicional, pero afortunadamente, la satisfacción de restricciones facilita su formulación y representación, así como también el cálculo de criterios múltiples de optimización.

Veamos tres ejemplos de codificación para restricciones adicionales complejas:

1. Para limitar la duración máxima de un plan a 100 unidades de tiempo, la restricción sería: $S(Start) + 100 \geq E(End)$.

2. Para que al finalizar un plan, un determinado fluent Φ tenga un valor de 50.5, la restricción sería: $V_{actual}(\Phi, End) \geq 50.5$. Para que el mismo fluent Φ sea satisfecho antes de que comience la acción b, la restricción sería: $Time(\Phi, End) < S(b)$.

3. Una función de multi-optimización de criterios, podría codificarse de la siguiente forma: $min\ 3 * V_{actual}(\Phi_i, End) - 2 * V_{actual}(\Phi_j, End) + 4 * E(End)$.

8.5.3. Restricciones II: Situaciones de Bifurcación

Análogamente a la planificación POCL clásica, la bifurcación es utilizada para generar el espacio de búsqueda de los diferentes planes parciales que son generados cuando:

- soportan una condición (proposicional/fluent), en cuyo caso un plan parcial es creado para cada alternativa posible en el Sup de la acción;

- aparece una relación de exclusión mutua entre acciones, en cuyo caso se coloca una restricción que prevenga que dos acciones modifiquen simultáneamente la misma proposión/fluent (es lo que se conoce como interferencia de efectos). Cabe observar sin embargo, que esto no impide que dos acciones se solapen, obteniéndose así una solución más permisiva que la planificación clásica;

- aparece una amenaza, son creados dos planes parciales disyuntivos: uno para soportar el conflicto por promoción y otro para solventarlo por democión.

Las restricciones contenidas en el grupo Restricciones II, serán analizadas en dos grupos, y similarmente a las pertenecientes al grupo de Restricciones I, sólo requerirán ser satisfechas aquellas restricciones de bifurcación cuyas acciones pertenezcan al plan, $(\forall a : Inplan(a) = 1)$.

Grupo RII.1: Restricciones de bifurcación para la parte proposicional de las acciones

- Soportes: $Sup(p, a) = b_i \wedge Sup(p, a) \neq b_j \mid \forall b_i, bj(b_i \neq b_j)$ que soporta p para la acción a, representa todas las posibilidades para soportar p, una por cada acción b_i.

- Resolución de amenazas: asumiendo que $Sup(\neg p)$ contiene las acciones que eliminan p, entonces $\forall b_i \in Sup(\neg p)$, $(Time(b_i, \neg p) < Time(p, a)) \vee (Req_{end}(p, a) < Time(b_i, \neg p))$.

- Resolución de relaciones de exclusión mutua: asumiendo que $Sup(\neg p)$ contiene las acciones que eliminan p, entonces $\forall b_i \in Sup(\neg p)$, $Time(b_i, \neg p) \neq Time(p, a)$.

Grupo RII.2: Restricciones de bifurcación para la parte numérica de las acciones

- Soportes: $Sup(\Phi, a) = b_i \wedge Sup(\Phi, a) \neq b_j \mid \forall b_i, b_j \, (b_i \neq b_j)$ que soporta Φ para la acción a, representa todas las posibilidades de soporte (o actualización) del fluent Φ.

- Resolución de amenazas: se realiza por medio de disyunciones, para evitar la actualización Φ entre $Time(\Phi, a)$ y el $Req_{end}(\Phi, a)$ $\forall b_i$ que actualiza $\Phi \mid b_i \notin Sup(\Phi, a)$, $time(b_i, \Phi) < Time(\Phi, a) \vee Req_{end}(\Phi, a) < Time(b_i, a))$.

- Resolución de relaciones de exclusión mutua (mutex): previene que dos acciones actualicen simultáneamente el mismo fluent Φ: $\forall b_i$ que actualiza $\Phi \mid b_i \notin Sup(\Phi, a)$, $Time(b_i, \Phi) \neq Time(\Phi, a)$.

8.5.4. Heurísticas

Tal como hemos explicado en los capítulos precedentes, en la resolución del CSP se pueden aplicar *técnicas de consistencia* en etapa de pre-proceso y durante la búsqueda; y diferentes *heurísticas* que permiten guiar la búsqueda, tales como las heurísticas de ordenación de variables, de ordenación de valores y de ordenación de restricciones. Para este tipo de problemas no-binarios, nos centramos en el desarrollo de heurísticas y dado que la mayoría de los resolutores de CSP tienen implementadas las heurísticas de ordenación de variables y de ordenación de valores, en este trabajo hemos desarrollado dos heurísticas: medida de alcanzabilidad y selección de variables presentes en el plan.

Heurística de medida de alcanzabilidad

La heurística de medida de alcanzabilidad está incluida en la formulación del CSP y es calculada en la parte de pre-procesamiento. Consiste en calcular el tiempo más temprano de inicio de cada una de las acciones en un grafo de planificación relajado. Se basa en la construcción de un grafo de planificación relajado que toma en cuenta las condiciones proposicionales, similar a lo propuesto en [57, 38, 56, 66] y al que se le añaden las restricciones numéricas, partiendo con las proposiciones y fluents del estado inicial y añadiendo/modificando los tiempos y valores de las proposiciones y fluents al aplicar las acciones. Es una heurística del tipo admisible. El valor resultante de la heurística, tiene efecto en el límite inferior del intervalo de inicio y finalización de la acción.

Heurística de ordenación de variables presentes en el plan

La heurística de ordenación de variables presentes en el plan requiere ser desarrollada dentro del resolutor de CSP. El criterio de cómo se seleccionarán las variables es muy importante en un CSP especialmente cuando existen muchas variables. En vez de utilizar una ordenación de variables a ciegas, esta heurística le da prioridad a los soportes de las acciones que están presentes en el plan. Se decide en tiempo de ejecución del CSP, seleccionando todas las variables de soporte de la acción *End*. Entonces, cuando el resolutor CSP prueba un valor v_i para cada una de estas variables, asigna un valor para la acción que soportará el enlace causal. En ese momento, la lista es actualizada con los soportes que la acción añadida necesita. Si el resolutor CSP realiza retrocesos (backtracks) y reemplaza el valor v_i con uno v_j, la lista es actualizada como corresponde y todos los valores insertados para v_i son reemplazados por los de v_j, y el proceso comienza nuevamente.

8.6. ClassP: una arquitectura para problemas $P_{P\&S}$

ClassP es la arquitectura propuesta que permite modelar y resolver problemas de planificación y scheduling. El modelado que realiza ClassP es nuestra formulación CSP para planificación POCL con todas las variables y restricciones mencionadas anteriormente, donde la formulación propuesta es codificada en función al modo de resolución seleccionado (scheduler, pseudo-planner y planificador), siguiendo la semántica de Choco[74] e invocando a Choco en la resolución.

ClassP fue desarrollado utilizando Java [113] y la librería de Choco. Así, la codificación CSP es modificada por ClassP para hacer tres modos de trabajo diferentes: planificador, pseudo-planner y scheduler, donde las variables $InPlan(a)$ tienen un rol importante para el resolutor de CSP, como se especifica a continuación:

- Si todas las variables $InPlan(a)$ son inicializadas en un valor 0 o 1 y las variables de soporte proposicional $Sup(p, a)$ y soporte numérico $Sup(\Phi, a)$ contienen un único valor, el resolutor de CSP actuará como un scheduler ya que su trabajo se limitará a asignar valores al resto de las variables (tiempo y recursos numéricos).

- Por el contrario, si ninguna variable $InPlan(a)$, $Sup(p, a)$ y $Sup(\Phi, a)$ es inicializada, el resolutor de CSP actuará como un planificador numérico (realizará planificación y scheduling), lo que implica seleccionar las acciones a ser incluidas en el plan (colocar a cada acción a que pertenezca al plan su $InPlan(a) = 1$) y asignar valores de tiempo y de recursos numéricos.

- En un término intermedio estará el asignar sólo a algunas variables su presencia obligatoria en el plan ($InPlan(a) = 1$), dejándole al resolutor de CSP conseguir los soportes válidos para dichas acciones obligatorias, permitiendo así diferentes grados de compromiso en función del número de variables asignadas.

ClassP está conformado por las siguientes agrupaciones de clases:

1. *Acciones*: son almacenadas en un vector, para cada una de las acciones almacena un registro con los siguientes datos: id, nombre, intervalo de duración, y una serie de vectores donde almacena estructuras de condiciones proposicionales, efectos positivos y negativos proposicionales, condiciones numéricas y efectos numéricos, y la heurística. Estos vectores hacen referencia a las estructuras de Proposiciones y Fluents.

2. *Proposiciones*: son almacenadas en un vector, en una estructura de registro que contiene: su id, nombre y un conjunto de vectores que apuntan a las acciones que tienen a la proposición como condición, efecto positivo o efecto negativo. Posee también un vector para las heurísticas.

3. *Fluents*: estructura que almacena las variables numéricas (fluents). En cada registro del fluent se almacena: su id, nombre y los vectores que referencia al vector de acciones con aquellas que poseen condiciones numéricas y/o efectos numéricos.

4. *Tools I/O*: Está conformado por una serie de conjuntos que almacenan los diferentes símbolos terminales y pre-símbolos que componen un problema de planificación instanciado por MIPS-XXL, así como los métodos para realizar la lectura y escritura de ficheros.

5. *Heurísticas*: son implementadas en las clases QueueOfPriority, Comprobator y Heuristics, para actualizar los valores de la heurística de análisis de alcanzabilidad.

6. *Principal*, es la aplicación que invoca a las demás, permitiéndole al usuario elegir diversas funcionalidades, utilizando para ello una serie de parámetros de entrada.

Así, la entrada para ClassP será un problema de planificación (dominio y problema) codificado en PDDL instanciado por MIPS-XXL [46]; donde las restricciones adicionales que no son soportadas por PDDL son añadidas posteriormente. Si la forma de trabajo que se requiere en la resolución es de scheduler o de pseudo-planner, debe incluirse en la entrada de datos un fichero con el plan completo (en caso de scheduler) o las acciones que deban ser obligatorias (para el caso de modo pseudo-planner).

En el Apéndice A, se muestra la resolución de problemas de dominios propuestos en la IPC (siglas en inglés: International Planning Competition) y de dominios ad hoc por los planificadores numéricos MIPS-XXL [46] y LPG-TD [65] en relación al lenguaje de modelado propuesto codificado en JAVA y su resolución realizada con ClassP, para demostrar que ClassP proporciona resultados correctos.

8.7. Ejemplo de aplicación

En esta sección presentamos un ejemplo de aplicación que muestra el alcance de la formulación propuesta. Empezaremos por un ejemplo sencillo al cual se le irán incrementando gradualmente las extensiones propuestas anteriormente que permitirán modelar restricciones más complejas. Nuestro principal interés es mostrar la expresividad de la codificación presentada y verificar la validez del plan obtenido.

Así, el problema de aplicación elegido es un problema de logística llamado Zeno Travel, de la Competición Internacional de Planificación (IPC-2002 [82]) el cual consiste en transportar por vía aérea a pasajeros y/o trasladar aviones entre diferentes ciudades en el menor tiempo posible.

Para el ejemplo de aplicación que proponemos, el escenario estará compuesto

por dos pasajeros (*person*1 y *person*2), dos aviones (*plane*1 y *plane*2) y tres ciudades (*city*0, *city*1 y *city*2). Se pueden aplicar tres operadores: *fly* (volar), *board* (embarcar) y *debark* (desembarcar) (ver Tabla 8.1). Similar a problemas del mundo real, no todas las aerolíneas vuelan a todas las ciudades, por lo que el avión *plane*2 sólo volará entre las ciudades *city*0 y *city*2 por ser una línea local, mientras que el avión *plane*1 puede volar a las tres ciudades. Asumimos que la duración es entera para todos los operadores[2]. De esta manera, al instanciar los operadores, todas las acciones *board* tendrán una duración de 3 unidades de tiempo; todas las acciones *debark* tendrán una duración de 2 unidades de tiempo y la acción de fly tendrá las duraciones de 7, 10 y 12, dependiendo las ciudades donde sea llevado a cabo.

Tabla 8.1: Operadores del dominio Zeno Travel utilizados para el Ejemplo 8.4. $S()$ y $E()$ determinan el punto de inicio y final del operador.

Operador	Parámetros	Condiciones	Efectos
fly ($volar$)	?$plane$?$origin$?$destination$	$S()..S() : at(?plane, ?origin)$	$S()..S() : \neg at(?plane, ?origin)$ $E()..E() : at(?plane, ?destination)$
$board$ ($embarcar$)	?$person$?$plane$?$city$	$S()..S() : at(?person, ?city)$ $S()..E() : at(?plane, ?city)$	$S()..S() : \neg at(?person, ?city)$ $E()..E() : in(?person, ?plane)$
$debark$ ($desembarcar$)	?$plane$?$person$?$city$	$S()..S() : at(?person, ?plane)$ $S()..E() : at(?plane, ?city)$	$S()..S() : \neg in(?person, ?plane)$ $E()..E() : at(?person, ?destination)$

Los estados inicial y objetivo son mostrados en la Figura 8.4. Los operadores con sus parámetros que se pueden aplicar siguiendo la plantilla (esquema de operadores utilizado en planificación) se muestran en la Tabla 8.1. También, para cada operador, se incluyen sus condiciones, sus efectos y el tiempo $S()$ y/o $E()$ en el cual son requeridos cada una de las condiciones y generados cada uno de los efectos, de dicha acción.

[2]Ello es debido a que el resolutor Choco (donde se implementó la propuesta) no manejaba dominios reales.

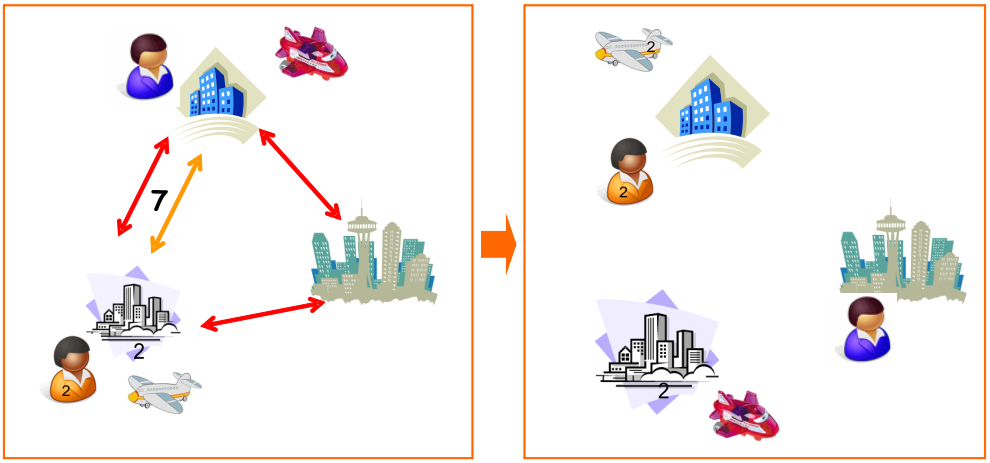


Figura 8.4: Estado Inicial (izquierda) y Estado Objetivo (derecha) para el problema de aplicación. Las flechas rojas indican donde puede volar el avión1 y la flecha amarilla donde puede volar el avión2, el número entre las flechas indica la duración de la acción *fly* entre cada par de ciudades.

A efectos de mostrar las diferentes características de nuestra propuesta, las restricciones del modelo extendido fueron añadidas de forma incremental, con lo cual se generaron cinco problemas diferentes (a los que llamaremos test) que serán descritos a continuación:

- Test 1: Es la resolución del problema básico.
- Test 2: Contiene restricciones sobre condiciones y efectos fuera del intervalo de duración de la acción.
- Test 3: Contiene tiempos máximo de finalización en todo el plan y en la ejecución de acciones.
- Test 4: Contiene restricciones entre acciones.
- Test 5: Contiene variables numéricas y ventanas temporales para su realización.

La solución de cada uno de los test fue realizada utilizando nuestro resolutor de CSP: ClassP, implementado en java que utiliza las librería de Choco [74] en las opciones: a) planificador (realización de un plan schedulado completo) y en b) scheduler (recibiendo un plan de entrada) que indicará qué acciones deben obligatoriamente

aparecer en el plan ($InPlan(a) = 1$), por lo que el trabajo del resolutor de CSP se limitó a fijar los instantes de tiempo para cada una de las variables y sus respectivos soportes.

Todas las pruebas fueron ejecutadas en un ordenador con procesador Pentium IV 2.6 GHz con 128 MB de RAM, limitando la búsqueda a 300 segundos. Se realizó una exploración completa del árbol de búsqueda, para encontrar el plan óptimo, y los resultados mostrados reflejan el uso o no de la heurística del tiempo mínimo de inicio de las acciones, propuesta en la sección anterior. Ambos modos utilizan la heurística del mínimo dominio (MinDomain) provista por la librería de Choco.

El Test 1, contiene el problema básico a resolver. Su plan óptimo tiene una duración de 29 unidades de tiempo y es mostrado en la Figura 8.5, donde puede observarse que contiene dos conjuntos de acciones independientes que se pueden realizar paralelamente. El conjunto de acciones más corto es para transportar al pasajero ($person2$) con el avion ($plane2$) desde la ciudad origen ($city2$) a la ciudad destino ($city0$) y el conjunto de acciones más largo para transportar al pasajero ($person1$) a la ciudad destino ($city1$) y dejar el avión ($plane1$) en la ciudad (city2).

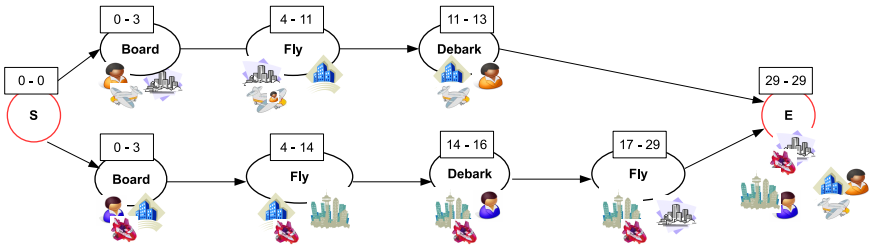


Figura 8.5: Test 1, plan óptimo para el problema de aplicación.

Para el Test 2, fueron añadidos condiciones y efectos complejos, comunes cuando se trata de transporte aéreo, como son que el pasajero debe estar cierto tiempo antes en el aeropuerto para realizar los chequeos de seguridad; el avión requiere de cierto tiempo para atravesar la pista del aeropuerto y una vez desembarcado el pasajero debe pasar por chequeos de inmigración. En este sentido, la condición $at(?person, ?city)$ del operador *board* debe hacerse efectiva 1 unidad de tiempo antes

de que la acción *board* se inicie. Adicionalmente, el efecto $at(?person, ?city)$ del operador *debark* y el efecto $at(?plane, ?city)$ del operador fly son generados 1 unidad de tiempo posterior a la ejecución de las acciones *debark* y fly respectivamente (i.e., $time_{before} = time_{after} = 1$). El plan óptimo para el Test 2 es de 32 unidades de tiempo y es mostrado en la Figura 8.6. Si estas restricciones fueran modeladas en la representación clásica de planificación, implicaría crear nuevos operadores para modelar los retrasos, pero con nuestra extensión propuesta sólo hay que añadir las siguientes restricciones:

- $Req_{start}(at(person_i, city_l), board(person_i, plane_k, city_l)) =$

$$S(board(person_i, plane_k, city_l)) - time_{before}$$

- **si** $Sup(at(plane_i, city_j), board(person_i, plane_k, city_j)) =$

$$debark(person_i, plane_l, city_j)$$

entonces

$$Time(at(person_i, city_j), board(person_i, plane_k, city_j)) =$$

$$E(debark(person_i, plane_l, city_j)) + time_{after}$$

- **si** $Sup(at(plane_i, city_j), a_k) = fly(plane_i, city_m, city_j)$ **entonces**

$$Time(at(plane_i, city_j), a_k) =$$

$$E(fly(plane_i, city_m, city_j)) + timeafter$$

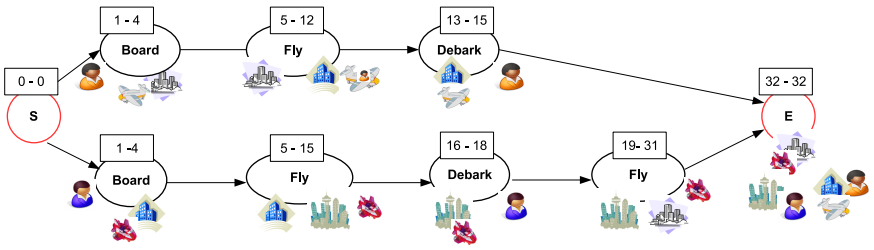


Figura 8.6: Test 2, plan óptimo para el problema de aplicación.

El Test 3, extiende los resultados del Test2 añadiéndole tiempos máximos de finalización en los objetivos (*deadlines*), forzando cambios en el plan. Para el Test 3a se incluyó la restricción de que el avión $plane2$ esté en la ciudad destino como máximo en 10 unidades de tiempo, lo cual requirió añadir la siguiente restricción:

- $Time(at(plane2, city0), End) \leq 10$.

La presencia de esta restricción hace que el pasajero $person2$ no pueda abordar el avión $plane2$ y deba viajar en el avión $plane1$, por lo que la duración total del plan (makespan) aumenta a 51 unidades de tiempo (ver Figura 8.7).

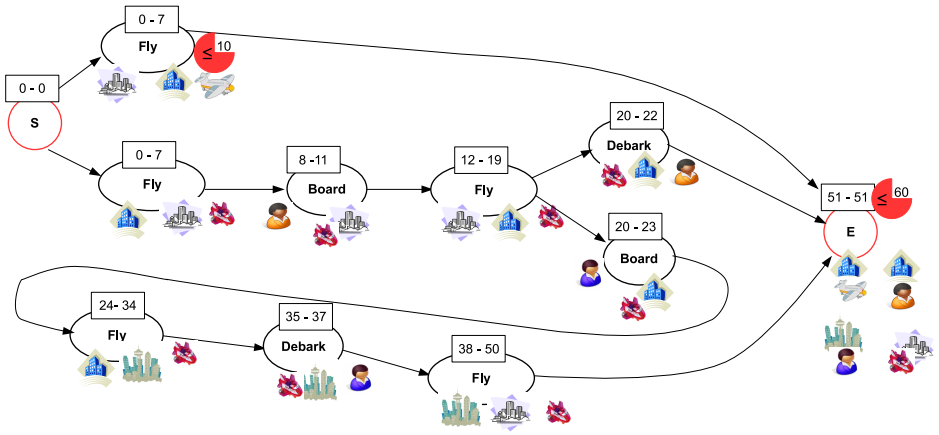


Figura 8.7: Test 3a, plan óptimo para el problema de aplicación, donde se colocan tiempos de finalización en recursos y en la duración total del plan.

Se realizó una prueba adicional que consistió en incluir una restricción sobre la duración total del plan a 50 unidades de tiempo:

- $End \leq 50$.

Esta restricción es muy estricta y hace que el problema no tenga solución, y lo presentamos como Test 3b.

Para el Test 4, algunas restricciones sobre *deadlines* de los Test 3a y 3b, son eliminadas, quedando sólo una restricción sobre la duración total del plan: ($End \leq 60$) y son añadidas restricciones entre acciones. Para ello se modelaron dos nuevas

restricciones: a) que una misma tripulación trabaje en dos aviones distintos, para lo cual fue añadida la restricción que el avión $plane2$ debe estar en la ciudad $city2$ antes que el avión $plane1$ despegue; y b) debe pasar un tiempo $time_{consecutive}$ entre dos vuelos consecutivos de un mismo avión, para su revisión preventiva y descanso de la tripulación. Así las restricciones incluidas en el modelo fueron:

- $E(fly(plane2, city2, city0)) + time_{before} \leq S(fly(plane1, city0, city_i))$
- $E(fly(plane_i, city_j, city_k)) + time_{consecutive} \leq S(fly(plane_i, city_k, city_l))$

El tiempo de espera está representado en las variables $time_{before} = 2$ y $time_{consecutive} = 5$. El tiempo de duración óptimo para este test es de 42 unidades de tiempo y su plan es mostrado en la Figura 8.8.

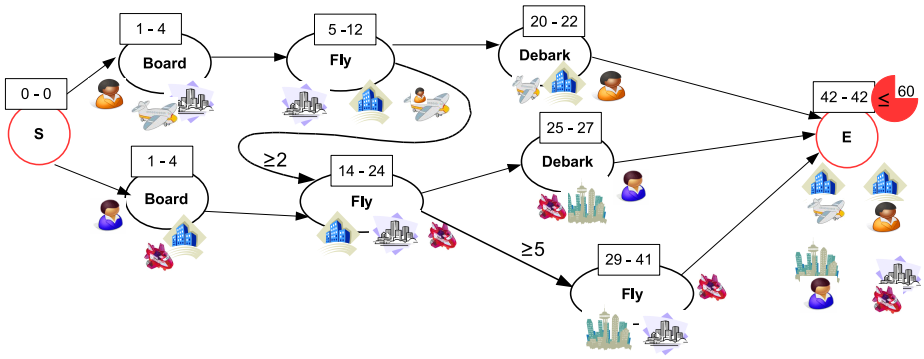


Figura 8.8: Test 4, plan óptimo para el problema de aplicación, donde se colocan restricciones de precedencia entre acciones y proposiciones.

Finalmente en el Test 5, al problema original planteado en Test 1 se le incluyó el manejo de recursos fungibles ($fluent$), como lo es el combustible ($fuel$) del avión, el cual sólo puede ser repuesto dentro de unos intervalos de tiempo (ventanas temporales) por lo que la codificación del CSP debe ahora incluir el manejo de este recurso $fuel$ en el operador de volar (fly) y debe añadirse un nuevo operador, la reposición de combustible ($refuel$). Todo ello hará que se generen las variables y restricciones mostradas en la Tabla 8.2. El tiempo de duración óptimo para el Test 5 es de 37

unidades de tiempo, ya que requiere la realización de un $refuel$ para el avión $plane1$, como puede ser observado en el plan mostrado en la Figura 8.9. Cabe destacar que con el manejo de recursos el factor de ramificación aumenta considerablemente lo que complica el proceso de búsqueda, lo que justifica el diseño y uso de técnicas de consistencia para problemas no-binarios con el fin de reducir el espacio de búsqueda.

Tabla 8.2: Variables y dominios requeridos en la codificación CSP, cuando es añadida la acción $refuel$ en el problema Zeno Travel de ejemplo, Test 5.

Variables y Dominios
$S(refuel(plane_i, city_j))) \in [0, \infty)$
$F(refuel(plane_i, city_j))) \in [0, \infty)$
$dur(refuel(plane1, city_j) \in [4, 5]$
$dur(refuel(plane2, city_j) \in [2, 3]$
$InPlan(refuel(plane_i, city_j)) \in [0, 1]$
$Sup(at(plane_i, city_j), refuel(plane_i, city_j)) \in \{fly(plane_i, city_k, city_j), Start\}$
$Sup(fuel(plane_i), fly(plane_i, city_j, city_k)) \in \{fly(plane_i, city_l, city_j), refuel(plane_i, city_j), Start\}$
$Time(at(plane_i, city_j)), refuel(plane_i, city_j)) \in [0, \infty)$
$Time(fuel(plane_i), fly(plane_i, city_j, city_k)) \in [0, \infty)$
$Req_{start}(at(plane_i, city_j), refuel(plane_i, city_j)) = S(refuel(plane_i, city_j))$
$Req_{end}(at(plane_i, city_j), refuel(plane_i, city_j)) = E(refuel(plane_i, city_j))$
$Req_{start}(fuel(plane_i), refuel(plane_i, city_j)) = S(refuel(plane_i, city_j))$
$Req_{end}(fuel(plane_i), refuel(plane_i, city_j)) = E(refuel(plane_i, city_j))$
$Req_{start}(fuel(plane_i), fly(plane_i, city_j, city_k)) = S(fly(plane_i, city_j, city_k))$
$Req_{end}(fuel(plane_i), fly(plane_i, city_j, city_k)) = E(fly(plane_i, city_j, city_k))$

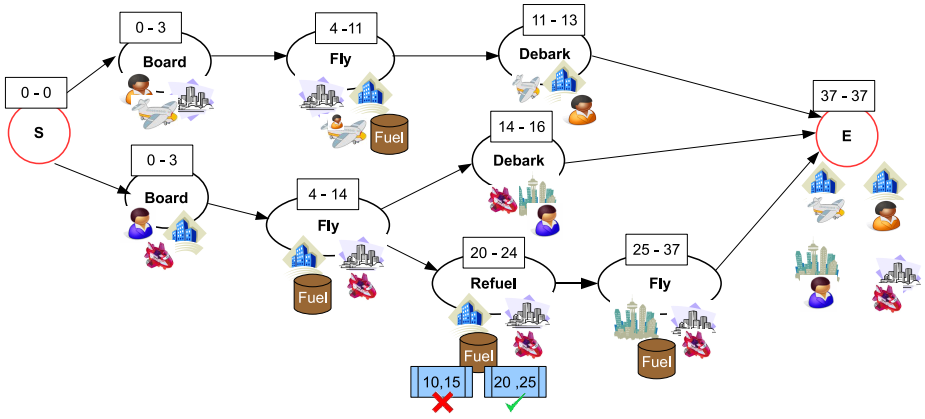


Figura 8.9: Test 5, plan óptimo para el problema de aplicación, donde se añaden recursos numéricos como el $fuel$, para ambos aviones.

Adicional a las restricciones de inicio S, finalización E, duración Dur, tiempo $Time$, inicio de requerimiento Req_{start} y finalización de requerimiento Req_{end} para la acción $refuel$, son añadidas las restricciones de las ventanas temporales para su realización, las cuales se codifican de la siguiente forma:

- **si** $((InPlan(refuel(plane_i, city_j)) = 1$ **entonces**

$$10 \leq S(refuel(plane_i, city_j)) \leq E(refuel(plane_i, city_j)) \leq 15$$

o

$$20 \leq S(refuel(plane_i, city_j)) \leq E(refuel(plane_i, city_j)) \leq 25$$

.

Tabla 8.3: Resultados de los cinco Test para el Ejemplo 8.4, ejecutado ClassP en modo Planificador aplicando las heurísticas MinDomain y Medida de alcanzabilidad.

Test	MinDomain			MinDomain+ Medida de alcanzabilidad		
	tiempo	Nodos	makespan	tiempo	Nodos	makespan
Test 1	44.7	1611	39	110.6	1611	39
	44.9	2103	29	111.5	2103	29
	45.2	3148	-	112.5	3139	-
Test 2	27.3	1531	42	28.2	1531	42
	27.5	2016	32	28.5	2016	32
	27.9	3153	-	28.9	3144	-
Test 3a	125.7	82839	55	114.7	82839	55
	125.7	82866	51	114.7	82866	51
	130.8	89717	-	119.6	89717	-
Test 3b	15.6	18347	∄	15	18335	∄
Test 4	65.5	71149	42	61.4	71149	42
	65.9	72279	-	61.8	72279	-
Test 5	66.7	85927	33	65.4	84715	33
	116.2	182784	-	95.9	147879	-

En cuanto a la eficiencia de la heurística del dominio mínimo (MinDomain), cuando el resolutor de CSP trabaja en modo planificador, se puede notar que su

utilidad es evidente cuando se incrementa la complejidad de las restricciones, como sucede a partir del Test 3 hasta el Test 5 (ver Tabla 8.3), donde el uso de esta heurística permite alcanzar en menos tiempo las soluciones y se disminuye así la cantidad de de nodos visitados. La cantidad de nodos visitados y el tiempo de cómputo también puede disminuirse si se aplicaran técnicas de consistencia. De ser aplicadas en etapa de pre-proceso, deberían ser técnicas de consistencia no-binarias. De ser aplicadas durante la búsqueda, se podrían aplicar las técnicas de arco-consistencia presentadas en los Capítulos 2 y 3 cuando se le asigne valor a la variable *inplan*, como por ejemplo en el caso de las restricciones tipo I.

Cuando el resolutor de CSP trabaja en modo scheduler, sólo se observa una leve disminución de nodos visitados en el Test 5, manteniéndose igual el tiempo de cómputo, por lo que se infiere que en este caso su uso no se hace necesario (ver Tabla 8.4).

Tabla 8.4: Resultados de los cinco Test para el Ejemplo 8.4, ejecutado ClassP en modo Scheduler aplicando las heurísticas MinDomain y Medida de alcanzabilidad.

	MinDomain			MinDomain+ Medida de alcanzabilidad		
Test	tiempo	Nodos	makespan	tiempo	Nodos	makespan
Test 1	0.1	12	29	0.1	12	29
	0.1	12	-	0.1	12	-
Test 2	0.1	12	32	0.1	12	32
	0.1	12	-	0.1	12	-
	0.2	43	55	0.2	43	55
Test 3a	0.3	70	51	0.3	70	51
	0.4	118	-	0.4	118	-
Test 3b	0.5	92	∄	0.3	92	∄
Test 4	0.1	20	42	0.1	20	42
	0.1	20	-	0.1	20	-
Test 5	0.1	20	33	0.1	20	33
	0.1	22	-	0.1	20	-

8.8. Conclusiones

En este capítulo hemos presentado una formulación general para codificar problemas de planificación y scheduling, independiente del dominio, utilizando POCL y CSP que incluye el modelo de acciones extendido propuesto, basada en la propuesta por Vidal y Geffner, CPT; y a ClassP una arquitectura para el modelado y resolución de este tipo de problemas que trabaja en tres modalidades: scheduler, pseudo-planner y planificador, obteniéndose los mejores resultados en la modalidad de scheduler, donde realizó muy eficientemente la asignación de recursos numéricos y temporales.

Capítulo 9

Conclusiones y Trabajos Futuros

9.1. Contribuciones destacadas

Este trabajo de tesis doctoral se centró principalmente en el desarrollo de técnicas de consistencia para Problemas de Satisfacción de Restricciones. No obstante, el trabajo consta de dos partes claramente diferenciadas:

- por un lado el diseño y la implementación de algoritmos de consistencia y búsqueda, para resolver CSPs binarios normalizados y no-normalizados.

- por otro lado la modelización y resolución de CSPs no-binarios, para resolver problemas de planificación y scheduling.

En una primera parte, (Capítulo 3) analizamos y reformulamos los algoritmos de arco-consistencia existentes en la literatura, con la finalidad de reducir el número de chequeos, cantidad de propagaciones y el tiempo de cómputo en CSPs binarios normalizados, proponiendo los algoritmos: AC3-OP [9, 11], AC4-OP [10] y AC2001-OP [15]. Los algoritmos AC3-OP y AC2001-OP son algoritmos de arco-consistencia de grano-grueso, que reducen el número de propagaciones en restricciones de orden y con ello reducen el tiempo de cómputo y la cantidad de chequeos. El algoritmo AC4-OP, es un algoritmo de arco-consistencia de grano-fino, que añade bidireccionalidad al proceso de chequeos y evita propagaciones innecesarias, logrando disminuir hasta en un 50 % el número de chequeos.

Basados en la propuesta realizada por Bessiere en [101], extendimos los algoritmos de arco-consistencia existentes para que procesaran CSPs binarios no-normalizados, proponiendo los algoritmos de arco-consistencia: AC3-NN [12] y AC4-OPNN [12, 6]. El algoritmo AC3-NN es un algoritmo de arco-consistencia de grano-grueso, con estructuras de datos muy ligeras, capaz de procesar rápidamente problemas binarios no-normalizados. El algoritmo AC4-OPNN, es un algoritmo de grano-fino que busca todos los soportes, realiza chequeos bidireccionalmente y evita propagaciones innecesarias, para problemas binarios no-normalizados. Así mismo, con la propuesta del algoritmo AC3NH [14], demostramos que el proceso de normalización en problemas binarios no-normalizados no tiene un coste trivial, por lo que se propone trabajar con el problema original y con algoritmos de consistencia que procesen dichos problemas no-normalizados.

Para esta tipología de problemas no-normalizados, proponemos (Capítulo 4) nuevos algoritmos que alcanzan la 2-consistencia: 2-C3 [7], 2-C3OP [8, 13], 2-C3OPL [16, 17], 2-C4 [19] y 2-C6, los cuales son capaces de realizar más poda, reducir tiempo de cómputo, chequeos y propagaciones con respecto a los algoritmos de arco-consistencia existentes en la literatura. Los algoritmos de 2-consistencia propuestos abarcan las dos modalidades de propagación: grano-fino (2-C4 y 2-C6) y grano-grueso (2-C3, 2-C3OP y 2-C3OPL); realizan los chequeos bidireccionalmente (2-C3OP, 2-C3OPL y 2-C4); almacenan los soportes encontrados (2-C3OPL y 2-C4) y limitan la búsqueda a un sólo soporte (2-C3, 2-C3OP, 2-C3OPL y 2-C6) o a todos los soportes (2-C4).

En cuanto a la resolución de CSPs binarios no-normalizados, proponemos dos técnicas de búsqueda (Capítulo 5): BLS (búsqueda heurística independiente del dominio) y SchTrains (búsqueda guiada, dependiente del dominio para la planificación de horarios ferroviarios) [18]. El desempeño de las técnicas de arco-consistencia, 2-consistencia y búsqueda propuestas se evaluó empíricamente con los algoritmos existentes en la literatura tanto en problemas: aleatorios, benchmarks y reales (Capítulos 6 y 7), donde cada algoritmo mejoró su comportamiento en comparación a su algoritmo base. Además, aplicamos las técnicas propuestas a un problema real, el problema de la planificación de horarios ferroviarios, que fue modelado como un CSP binario para su posterior resolución, donde los algoritmos AC2001-OP, 2-C3OPL y SchTrains fueron los que mostraron mejor comportamiento.

Para la resolución de problemas que involucran planificación y scheduling, proponemos el modelado de CSPs no-binarios (Capítulo 8), una extensión de CTP [121], logrando codificar restricciones reales más complejas, que extienden el lenguaje PDDL 2.1. En este sentido presentamos a ClassP, el cual permite modelar el problema y tres formas de resolución: planificador, pseudo-planificador y scheduler [59, 4, 55]. El modelado propuesto también permite trabajar en entornos distribuidos [111, 112], realizar reparación de planes cuyos objetivos no se han alcanzado [5, 3], ampliando lo propuesto en [48, 110, 119, 118], así como también la posibilidad de aplicar los algoritmos de consistencia propuestos.

Por lo tanto, podemos resumir el trabajo realizado en los siguientes puntos:

- Propuesta de algoritmos de arco-consistencia para CSPs binarios normalizados con restricciones de orden: AC3-OP y AC2001-OP.
- Propuesta de algoritmo de arco-consistencia para CSPs binarios normalizados: AC4-OP.
- Propuesta de algoritmos de arco-consistencia para CSPs binarios no-normalizados: AC3-NN y AC4-OPNN.
- Propuesta de algoritmo de normalización híbrida para CSPs binarios no-normalizados: AC3NH.
- Propuesta de algoritmos de 2-consistencia para CSPs binarios no-normalizados: 2-C3, 2-C3OP, 2-C3OPL, 2-C4 y 2-C6.
- Diseño y desarrollo de un nuevo algoritmo de búsqueda heurístico independiente del dominio para CSPs binarios no-normalizados: BLS.
- Diseño y desarrollo de un nuevo algoritmo de búsqueda dependiente del dominio de planificación de horarios ferroviarios para CSPs binarios no-normalizados: SchTrains.
- Propuesta de extensión de lenguaje de modelado y codificación de CSPs no-binarios para problemas de planificación y scheduling: ClassP.
- Problemas de aplicación, evaluación experimental y comparativa con algoritmos existentes en la literatura para la resolución de CSPs.

9.2. Líneas Futuras de Desarrollo

El hecho de proponer algoritmos eficientes que alcancen la arco-consistencia es un punto central en la comunidad que investiga el razonamiento con restricciones, por lo que el trabajo desarrollado en esta tesis permitirá abrir nuevas líneas futuras de investigación, tales como:

- La posibilidad de generar nuevos algoritmos de consistencia y de búsqueda que permitan resolver grandes problemas no-normalizados de forma más eficiente, como el desarrollo de un algoritmo de grano-fino que almacene un sólo soporte con chequeo bidireccional en una estructura tipo Last, que consistiría en añadirle la bidireccionalidad a 2-C6 y recordar los soportes encontrados con verificación de su existencia previo a la realización de chequeos, utilizando como base AC7 [30] y las propuestas presentadas en 2-C4 [19] y 2-C3OP [8].

- El extender BLS para que realice el proceso de búsqueda con cualquier algoritmo de arco-consistencia y 2-consistencia que almacene soportes encontrados (por ejemplo: AC2001/3.1, AC2001-OP, AC4-OP, AC4-OPNN y AC7) y verificar su comportamiento.

- El diseño y desarrollo de algoritmos -al estilo MAC- para el mantenimiento de la 2-consistencia durante la búsqueda.

- El diseño de nuevos algoritmos que alcancen la 2-consistencia para CSPs continuos, como 2-consistencia de borde o también algoritmos que extiendan el chequeo de consistencia a una tercera variable como, 2-consistencia de senda.

- Otra posibilidad de investigación sería la de manejar el proceso de consistencia (arco-consistencia y 2-consistencia) en forma distribuida, utilizando las técnicas de partición de CSPs propuestas en [1], y con ello lograr trabajar con problemas más grandes y/o reducir el tiempo de cómputo.

- También queda abierto para investigación el modelado y resolución CSPs de otros problemas reales, como el problema del transporte marítimo y la generación de nuevas técnicas de consistencia para CSPs no-binarios.

9.3. Publicaciones Relacionadas con la Tesis Doctoral

En esta sección presentamos las publicaciones generadas en el marco de la tesis doctoral. Estas publicaciones las hemos clasificado en: artículos en revista listadas en JCR, congresos internacionales y congresos nacionales.

Artículos en Revistas listadas en JCR

- M. Arangú, M.A. Salido. *A Fine-Grained Arc-Consistency Algorithm for Non-Normalized Constraint Satisfaction Problems.* International Journal of Applied Mathematics and Computer Science. Vol 21(4), to appear, 2012.
  (JCR-2010: 0.794).

- M. Arangú, M.A. Salido, F. Barber. *A Filtering Technique to Achieve 2-consistency in Constraint Satisfaction Problems.* International Journal of Innovative Computing, Information and Control. Vol 8(4), to appear, 2012.
  (JCR-2010: 1.664).

- A. Garrido, M. Arangú, E. Onaindía. *A Constraint Programming Formulation for Planning: from Plan Scheduling to Plan Generation* Journal of Scheduling. Springer. Vol 12, pp: 227-256, 2009.
  (JCR: 1.265).

- O. Sapena, E. Onaindía, A. Garrido, M. Arangú. *A Distributed CSP Approach for Collaborative Planning Systems.* Engineering Applications of Artificial Intelligence. Elsevier. Vol 21(5), pp: 698-709, 2008.
  (JCR: 1.397).

Congresos Internacionales

- M. Arangú, M.A. Salido, F. Barber. *AC2001-OP: An Arc-Consistency Algorithm for Constraint Satisfaction Problems.* 23rd International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA-AIE-2010). LNAI 6098. Vol 3, pp: 219-228, 2010.
  (CSC: top 5 in Industrial Engineering Conference Ranking. ERA2010: C).

- M. Arangú, M.A. Salido, F. Barber. *A Filtering Technique for the Railway Scheduling Problem*. 20th International Conference on Automated Planning and Scheduling: Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (ICAPS-COPLAS 2010). pp: 68-78, 2010.
  (ERA2010: B).

- M. Arangú, M.A. Salido, F. Barber. *Extending arc-consistency algorithms for Non-Normalized CSPs*. 29th SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence (AI-2009). Research and Development in Intelligent Systems XXVI. Incorporating Applications and Innovations in Intelligent Systems XVII. pp: 311-316, 2009.
  (CORE C).

- M. Arangú, M.A. Salido, F. Barber. *2-C3OP: An Improved version of 2-Consistency*. 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009). pp: 344-348, 2009.
  (CSC Ranking: 0.74, CORE B).

- M. Arangú, M.A. Salido, F. Barber. *A Filtering Technique for Non-normalized CSPs*. 15th International Conference on Principles and Practice of Constraint Programming: Doctoral Programme (CP-DP 09). pp: 1-6, 2009.

- M. Arangú, M.A. Salido, F. Barber. *2-C3: From Arc-Consistency to 2-Consistency*. 8th Symposium on Abstraction, Reformulation and Approximation (SARA2009). pp: 184-189, 2009.
  (CORE A).

- M. Arangú, A. Garrido, E. Onaindía. *A general technique for plan repair*. 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008). IEEE Computer Society Press, Vol. 1, pp: 515-518, 2008.
  (CORE B).

- M. Arangú. *Cooperation between Planning and Scheduling for complex problem resolution*. 11th Ibero-American Conference on Artificial Intelligence: Workshop on Planning and Scheduling and Constraint Satisfaction (IBERAMIA-CSPS-08). pp: 19-32, 2008.

- O. Sapena, E. Onaindía, A. Garrido, M. Arangú. *A Distributed CSP Approach for Solving Multi-agent Problems.* 17th International Conference on Automated Planning & Scheduling and 13th International Conference on Principles and Practice of Constraint Programming: Joint Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (CP/ICAPS 2007-COPLAS'07). pp: 68-75, 2007.
  (CORE B).

- A. Garrido, E. Onaindía, M. Arangú. *Using Constraint Programming to Model Complex Plans in an Integrated Approach for Planning and Scheduling.* 25th Workshop of the UK Planning and Scheduling. Special Interest Group (PlanSIG2006). pp: 137-144, 2006.

Congresos Nacionales

- M. Arangú, M.A. Salido, F. Barber. *Técnicas de Consistencia para Problemas de Satisfacción de Restricciones.* VI Jornadas de Investigación y Postgrado del Decanato de Ciencias y Tecnología. Universidad Centroccidental L̈isandro Alvarado: 2010.

- M. Arangú, M.A. Salido, F. Barber. *Optimizing a fine-grained arc-consistency algorithm.* XIII Conferencia de la Asociación Española para la Inteligencia Artificial y Jornadas de Transferencia Tecnológica de Inteligencia Artificial (CAEPIA-TTIA 2009). pp: 201-210, 2009.
  (CSC Ranking: 0.55).

- M. Arangú, M.A. Salido, F. Barber. *Changing Propagations in Arc-Consistency Algorithm for Arithmetic Constraints.* CAEPIA 2009: Workshop on Planning, Scheduling and Constraint Satisfaction (CSPS-09). pp: 8-20, 2009.
  (CSC Ranking: 0.55).

- M. Arangú, M.A. Salido, F. Barber *Normalizando CSPs no-normalizados: un enfoque híbrido.* CAEPIA 2009: Workshop on Planning, Scheduling and Constraint Satisfaction (CSPS-09), pp: 57-68, 2009.
  (CSC Ranking: 0.55).

- M. Arangú, M.A. Salido, F. Barber. *AC3-OP: An Arc-Consistency Algorithm for Arithmetic Constraints*. 12th International Congress of the Catalan Association of Artificial Intelligence (CCIA 09). Vol. 202, pp: 293-300, 2009.
- Marlene Arangú, E. Onaindía, A. Garrido. *A Constraint Programming based approach for Planning and Scheduling Problems*. Problem Solving and Applications in User-Centric Technologies Seminar. Universidad Carlos III de Madrid. 2007.

Apéndice A

Resultados de la evaluación del lenguaje de modelado

A.1. Introducción

La implementación realizada con ClassP, presentada en el Capítulo 8, se enfocó a verificar la validez de la codificación y de los planes resultantes para problemas de Planificación y Scheduling, desestimando la eficiencia de ClassP en relación a otros planificadores. En tal sentido se realizaron pruebas en diferentes dominios tomados de la Competición Internacional de Planificación IPC [82] y sobre otros dominios formulados ad hoc, en los que diferenciamos dominios completamente proposicionales, dominios completamente numéricos o una combinación de ambos. Los planes proporcionados por ClassP fueron validados con respecto a los planes proporcionados por los planificadores numéricos MIPS-XXL [46] y LPG-TD [65].

Debido a que la codificación completa del lenguaje de modelado propuesto en el Capítulo 8 es muy extensa (tanto del problema como de la solución), para cada uno de los dominios se muestra una breve descripción del problema; los estados inicial y objetivo del problema, un plan solución obtenido de un planificador (LPG-TD o MIPS-XXL) y la solución CSP obtenida con ClassP (un subconjunto de las variables a entre las que se pueden incluir: $S(a)$, $E(a)$ (correspondientes al inicio y finalización de la acción a) y la variable $InPlan(a)$.

A.2. Dominio ad hoc: Adaptaplan

Adaptaplan es un problema de planificación desarrollado para el proyecto de Rutas de Aprendizaje[61, 58]. Su complejidad radica en que es 100 % numérico, por lo que planificadores que se basen, en los predicados proposicionales de las acciones para la obtención de los soportes, no pueden resolver problemas de este tipo. El planificador LPG-TD, -que utilizamos como base para la mayoría de nuestras pruebas-, no pudo obtener solución, por lo que utilizamos como segunda opción el planificador MIPS-XXL para contrastar nuestros resultados.

Tabla A.1: Problema Adaptaplan

Estado Inicial	**Estado Objetivo**
(:init (= (concepto1) 0) (= (concepto2) 0) (= (concepto3) 0) (= (concepto4) 0) (= (concepto5) 0) (= (concepto6) 0) (= (concepto7) 0) (= (conceptoPrev1) 25) (= (conceptoPrev2) 25) (= (repActividad1) 0) (= (repActividad2) 0) (= (repActividad3) 0) (= (repActividad4) 0) (= (repActividad5) 0) (= (repActividad6) 0) (= (repActividad7) 0) (= (repActividad8) 0) (= (repActividad9) 0) (= (repActividad10) 0) (= (repActividad11) 0) (= (repActividad12) 0) (= (repActividad13) 0) (= (repActividad14) 0) (= (totalResourceCost) 0) (= (penalty) 0))	(:goal (and (>= (CONCEPTO7) 100.00)))

A.3. Dominio DriverLogs

DriverLogs es una variación del dominio de logística. Pertenece a los problemas benchmarks de la IPC-2002. Se trata de un problema proposicional. Trata de un

Tabla A.2: Plan solución generado por el planificador MIPS-XXL para el problema Adaptaplan mostrado en la Tabla A.1

```
Plan-quality: 8.00

ff: found legal plan as follows

0.00: (ACTIVIDAD1) [2.00]
2.01: (ACTIVIDAD3VERBAL) [2.00]
4.02: (ACTIVIDAD4) [2.00]
6.03: (ACTIVIDAD6VERBIND) [5.00]
11.04: (ACTIVIDAD9) [1.50]
12.55: (ACTIVIDAD8) [2.00]
14.56: (ACTIVIDAD11IND) [2.00]
16.57: (ACTIVIDAD12BIBINDMEDIO) [1.00]

15.06
```

Tabla A.3: Solución CSP generada por ClassP utilizando Choco-CP para el problema Adaptaplan mostrado en la Tabla A.1

solve => 1 solutions	
212[+0] millis.	
3[+0] nodes	
Problem with integer variables	
Num Solutions: 1	
START: 0	INPLAN_START: 1
END: 16	INPLAN_END: 1
S_ACTIVIDAD12BIBINDMEDIO: 15	INPLAN_ACTIVIDAD12BIBINDMEDIO: 1
E_ACTIVIDAD12BIBINDMEDIO: 16	INPLAN_ACTIVIDAD11IND: 1
S_ACTIVIDAD11IND: 13	INPLAN_ACTIVIDAD9: 1
E_ACTIVIDAD11IND: 15	INPLAN_ACTIVIDAD8: 1
S_ACTIVIDAD9: 0	INPLAN_ACTIVIDAD6VERBIND: 1
E_ACTIVIDAD9: 2	INPLAN_ACTIVIDAD4: 1
S_ACTIVIDAD8: 11	INPLAN_ACTIVIDAD3VERBAL: 1
E_ACTIVIDAD8: 13	INPLAN_ACTIVIDAD1: 1
S_ACTIVIDAD6VERBIND: 6	
E_ACTIVIDAD6VERBIND: 11	
S_ACTIVIDAD4: 4	
E_ACTIVIDAD4: 6	
S_ACTIVIDAD3VERBAL: 2	
E_ACTIVIDAD3VERBAL: 4	
S_ACTIVIDAD1: 0	
E_ACTIVIDAD1: 2	

problema de reparto de paquetes entre diferentes ciudades utilizando camiones los cuales precisan de conductores que los manejen. Las personas y los camiones se desplazan por vías diferentes (caminos y carreteras, respectivamente).

Tabla A.4: Problema DriverLogs

Estado Inicial	**Estado Objetivo**
(:init (empty_TRUCK1) (at_TRUCK1_S0) (at_DRIVER1_S0))	(:goal (and (at_TRUCK1_S1) (at_DRIVER1_S1))) (:metric minimize (TOTAL-TIME))

Tabla A.5: Plan solución generado por el planificador LPG-TD para el problema DriverLogs mostrado en la Tabla A.4

```
; Version LPG-td-1.0
; Seed 82461552
; Command line: ./lpg-td-1.0 -o driverlogTimed.pddl -f pfile1 -quality
; Problem pfile1
; Time 0.76
; Plan generation time 0.06
; Search time 0.03
; Parsing time 0.03
; Mutex time 0.00
; MakeSpan 302.00

0.0003: (WALK DRIVER2 S2 P1-2) [79.0000]
79.0005: (WALK DRIVER2 P1-2 S1) [29.0000]
108.0007: (WALK DRIVER2 S1 P1-0) [43.0000]
151.0010: (WALK DRIVER2 P1-0 S0) [80.0000]
231.0013: (BOARD-TRUCK DRIVER2 TRUCK1 S0) [1.0000]
232.0015: (DRIVE-TRUCK TRUCK1 S0 S1 DRIVER2) [70.0000]
0.0018: (WALK DRIVER1 S2 P1-2) [79.0000]
79.0020: (WALK DRIVER1 P1-2 S1) [29.0000]
```

A.4. Dominio Rovers

Rovers es un dominio numérico de la IPC 2002. Consiste en un grupo de robots móviles (Rovers) que deben analizar muestras de rocas y tierra, tomar fotografías mientras se desplazan por la superficie de un planeta y posteriormente comunicar los resultados a la base (Lander).

Tabla A.6: Solución CSP generada por ClassP utilizando Choco-CP para el problema DriverLogs mostrado en la Tabla A.4

Solution #1 is found with 0[+245896] millis. 0[+895922] nodes; START: 0 END: 72 S_BOARD-TRUCK.DRIVER1.TRUCK1.S2: 0 E_BOARD-TRUCK.DRIVER1.TRUCK1.S2: 0 S_BOARD-TRUCK.DRIVER1.TRUCK1.S1: 0 E_BOARD-TRUCK.DRIVER1.TRUCK1.S1: 0 S_DRIVE-TRUCK.TRUCK1.S1.S0.DRIVER1: 0 E_DRIVE-TRUCK.TRUCK1.S1.S0.DRIVER1: 0 S_DRIVE-TRUCK.TRUCK1.S2.S0.DRIVER1: 0 E_DRIVE-TRUCK.TRUCK1.S2.S0.DRIVER1: 0 S_DRIVE-TRUCK.TRUCK1.S2.S1.DRIVER1: 0 E_DRIVE-TRUCK.TRUCK1.S2.S1.DRIVER1: 0 S_DRIVE-TRUCK.TRUCK1.S1.S2.DRIVER1: 0 E_DRIVE-TRUCK.TRUCK1.S1.S2.DRIVER1: 0 S_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S2: 0 E_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S2: 0 S_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S1: 71 E_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S1: 72 S_DRIVE-TRUCK.TRUCK1.S0.S1.DRIVER1: 1 E_DRIVE-TRUCK.TRUCK1.S0.S1.DRIVER1: 71 S_DRIVE-TRUCK.TRUCK1.S0.S2.DRIVER1: 0 E_DRIVE-TRUCK.TRUCK1.S0.S2.DRIVER1: 0 S_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S0: 0 E_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S0: 0 S_BOARD-TRUCK.DRIVER1.TRUCK1.S0: 0 E_BOARD-TRUCK.DRIVER1.TRUCK1.S0: 1	INPLAN_START: 1 INPLAN_END: 1 INPLAN_BOARD-TRUCK.DRIVER1.TRUCK1.S2: 0 INPLAN_BOARD-TRUCK.DRIVER1.TRUCK1.S1: 0 INPLAN_DRIVE-TRUCK.TRUCK1.S1.S0.DRIVER1: 0 INPLAN_DRIVE-TRUCK.TRUCK1.S2.S0.DRIVER1: 0 INPLAN_DRIVE-TRUCK.TRUCK1.S2.S1.DRIVER1: 0 INPLAN_DRIVE-TRUCK.TRUCK1.S1.S2.DRIVER1: 0 INPLAN_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S2: 0 INPLAN_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S1: 1 INPLAN_DRIVE-TRUCK.TRUCK1.S0.S1.DRIVER1: 1 INPLAN_DRIVE-TRUCK.TRUCK1.S0.S2.DRIVER1: 0 INPLAN_DISEMBARK-TRUCK.DRIVER1.TRUCK1.S0: 0 INPLAN_BOARD-TRUCK.DRIVER1.TRUCK1.S0: 1

Tabla A.7: Problema Rovers

Estado Inicial
*********Init Start Action predicado name=EMPTY_ROVER1STORE predicado name=AVAILABLE_ROVER1 predicado name=AT_ROVER1_WAYPOINT2 predicado name=EMPTY_ROVER0STORE predicado name=AVAILABLE_ROVER0 predicado name=AT_ROVER0_WAYPOINT3 predicado name=CHANNEL_FREE_GENERAL predicado name=AT_ROCK_SAMPLE_WAYPOINT3 predicado name=AT_SOIL_SAMPLE_WAYPOINT3 predicado name=AT_SOIL_SAMPLE_WAYPOINT2 predicado name=AT_ROCK_SAMPLE_WAYPOINT1 fluent name=ENERGY_ROVER1 =50 fluent name=ENERGY_ROVER0 =50
Estado Objetivo
*********Init End Action predicado name= COMMUNICATED_IMAGE_DATA_OBJ0_HIGH_RES predicado name= COMMUNICATED_ROCK_DATA_WAYPOINT1 predicado name= COMMUNICATED_SOIL_DATA_WAYPOINT3

Tabla A.8: Plan solución generado por el planificador LPG-TD para el problema Rovers mostrado en la Tabla A.7

```
; Version LPG-td-1.0
; Seed 41028539
; Command line: /home/marangu/Desktop/pruebas/lpg-td-1.0 -o CTRover.pddl -f pfile4 -quality
; Problem pfile4
; Time 0.26
; Plan generation time 0.05
; Search time 0.04
; Parsing time 0.01
; Mutex time 0.00
; MakeSpan 50.00

0.00: (SAMPLE_SOIL ROVER0 ROVER0STORE WAYPOINT3) [10.0]
10.00: (COMMUNICATE_SOIL_DATA ROVER0 GENERAL WAYPOINT3 WAYPOINT3 WAYPOINT2) [10.0]
0.00: (CALIBRATE ROVER1 CAMERA0 OBJE0 WAYPOINT2) [5.0000]
5.00: (TI ROVER1 WAYPOINT2 OBJE0 CAMERA0 HIGH_RES) [7.0]
12.00: (NAVIGATE ROVER1 WAYPOINT2 WAYPOINT1) [5.0000]
17.00: (SAMPLE_ROCK ROVER1 ROVER1STORE WAYPOINT1) [8.0]
25.00: (COMMUNICATE_ROCK_DATA ROVER1 GENERAL WAYPOINT1 WAYPOINT1 WAYPOINT2) [10.0]
35.00: (COMMUNICATE_IMAGE_DATA ROVER1 GENERAL OBJ0 HIGH_RES WAYPOINT1 WAYPOINT2) [15.0]
```

Solución CSP generada por ClassP utilizando Choco-CP para el problema Rovers mostrado en la Tabla A.7
solve => 1 solutions
981[+0] millis.
1088[+0] nodes
Problem with integer variables
Num Solutions: 1
START:0
END:83
INPLAN_START:1
INPLAN_END:1
INPLAN_NAVIGATE_ROVER0_WAYPOINT0_WAYPOINT3:0
INPLAN_NAVIGATE_ROVER0_WAYPOINT1_WAYPOINT3:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ1_HIGH_RES_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ1_COLOUR_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ0_HIGH_RES_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ0_COLOUR_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ1_HIGH_RES_WAYPOINT2_WAYPOINT0:1
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ1_COLOUR_WAYPOINT2_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ0_HIGH_RES_WAYPOINT2_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ0_COLOUR_WAYPOINT2_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ1_HIGH_RES_WAYPOINT3_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ1_COLOUR_WAYPOINT3_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ0_HIGH_RES_WAYPOINT3_WAYPOINT0:0
INPLAN_COMMUNICATE_IMAGE_DATA_ROVER0_GENERAL_OBJ0_COLOUR_WAYPOINT3_WAYPOINT0:0
INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT3_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT2_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT1_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT3_WAYPOINT2_WAYPOINT0:1
INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT2_WAYPOINT2_WAYPOINT0:0
INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT1_WAYPOINT2_WAYPOINT0:0
INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT3_WAYPOINT3_WAYPOINT0:0
INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT2_WAYPOINT3_WAYPOINT0:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Rovers mostrado en la Tabla A.7

INPLAN_COMMUNICATE_ROCK_DATA_ROVER0_GENERAL_WAYPOINT1_WAYPOINT3_WAYPOINT0:0
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT3_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT2_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT0_WAYPOINT1_WAYPOINT0:0
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT3_WAYPOINT2_WAYPOINT0:0
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT2_WAYPOINT2_WAYPOINT0:1
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT0_WAYPOINT2_WAYPOINT0:0
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT3_WAYPOINT3_WAYPOINT0:0
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT2_WAYPOINT3_WAYPOINT0:0
INPLAN_COMMUNICATE_SOIL_DATA_ROVER0_GENERAL_WAYPOINT0_WAYPOINT3_WAYPOINT0:0
INPLAN_TI_ROVER0_WAYPOINT0_OBJ0_CAMERA0_COLOUR:0
INPLAN_TI_ROVER0_WAYPOINT0_OBJ0_CAMERA0_HIGH_RES:0
INPLAN_TI_ROVER0_WAYPOINT1_OBJ0_CAMERA0_COLOUR:0
INPLAN_TI_ROVER0_WAYPOINT1_OBJ0_CAMERA0_HIGH_RES:0
INPLAN_TI_ROVER0_WAYPOINT2_OBJ0_CAMERA0_COLOUR:0
INPLAN_TI_ROVER0_WAYPOINT2_OBJ0_CAMERA0_HIGH_RES:0
INPLAN_TI_ROVER0_WAYPOINT3_OBJ0_CAMERA0_COLOUR:0
INPLAN_TI_ROVER0_WAYPOINT3_OBJ0_CAMERA0_HIGH_RES:0
INPLAN_TI_ROVER0_WAYPOINT0_OBJ1_CAMERA0_COLOUR:0
INPLAN_TI_ROVER0_WAYPOINT0_OBJ1_CAMERA0_HIGH_RES:0
INPLAN_TI_ROVER0_WAYPOINT1_OBJ1_CAMERA0_COLOUR:0
INPLAN_TI_ROVER0_WAYPOINT1_OBJ1_CAMERA0_HIGH_RES:0
INPLAN_TI_ROVER0_WAYPOINT2_OBJ1_CAMERA0_COLOUR:0
INPLAN_TI_ROVER0_WAYPOINT2_OBJ1_CAMERA0_HIGH_RES:0
INPLAN_TI_ROVER0_WAYPOINT3_OBJ1_CAMERA0_COLOUR:0
INPLAN_TI_ROVER0_WAYPOINT3_OBJ1_CAMERA0_HIGH_RES:1
INPLAN_CALIBRATE_ROVER0_CAMERA0_OBJ1_WAYPOINT0:0
INPLAN_CALIBRATE_ROVER0_CAMERA0_OBJ1_WAYPOINT1:0
INPLAN_CALIBRATE_ROVER0_CAMERA0_OBJ1_WAYPOINT2:0
INPLAN_CALIBRATE_ROVER0_CAMERA0_OBJ1_WAYPOINT3:1
INPLAN_DROP_ROVER0_ROVER0STORE:1
INPLAN_SAMPLE_ROCK_ROVER0_ROVER0STORE_WAYPOINT3:1
INPLAN_SAMPLE_ROCK_ROVER0_ROVER0STORE_WAYPOINT2:0
INPLAN_SAMPLE_ROCK_ROVER0_ROVER0STORE_WAYPOINT1:0
INPLAN_SAMPLE_SOIL_ROVER0_ROVER0STORE_WAYPOINT3:0
INPLAN_SAMPLE_SOIL_ROVER0_ROVER0STORE_WAYPOINT2:1
INPLAN_SAMPLE_SOIL_ROVER0_ROVER0STORE_WAYPOINT0:0
INPLAN_RECHARGE_ROVER0_WAYPOINT0:0
INPLAN_NAVIGATE_ROVER0_WAYPOINT2_WAYPOINT1:0
INPLAN_NAVIGATE_ROVER0_WAYPOINT1_WAYPOINT2:1
INPLAN_NAVIGATE_ROVER0_WAYPOINT3_WAYPOINT0:0
INPLAN_NAVIGATE_ROVER0_WAYPOINT3_WAYPOINT1:1

A.5. Dominio Satellite

El Dominio Satellite es de tipo proposicional y pertenece a la IPC-2002. Consiste en que un conjunto de satélites realicen una serie de fotografías en el espacio, utilizando los instrumentos que pueden requerir calibración. Su complejidad radica en que se manejan gran cantidad de acciones, así un plan de 25 tareas requiere buscar la solución en 756 acciones.

Tabla A.10: Problema Satellite

Estado Inicial
(:init (POINTING_SATELLITE3_GS2) (POWER_AVAIL_SATELLITE3) (POINTING_SATELLITE2_STAR6) (POWER_AVAIL_SATELLITE2) (POINTING_SATELLITE1_GS0) (POWER_AVAIL_SATELLITE1) (POINTING_SATELLITE0_STAR6) (POWER_AVAIL_SATELLITE0))
Estado Objetivo
(:goal (and (HAVE_IMAGE_P11_IMAGE2) (HAVE_IMAGE_P10_IMAGE0) (HAVE_IMAGE_P9_IMAGE3) (HAVE_IMAGE_P8_IMAGE0) (HAVE_IMAGE_STAR7_IMAGE0) (HAVE_IMAGE_STAR6_IMAGE1) (HAVE_IMAGE_PHENOMENON5_IMAGE0) (POINTING_SATELLITE2_PHENOMENON5) (POINTING_SATELLITE1_STAR1))

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

```
solve => 1 solutions
19691[+0] millis.
6012[+0] nodes
Problem with integer variables
Num Solutions: 1
START:0
END:105


INPLAN_START:1
INPLAN_END:1
```

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE3_PLANET10_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_STAR7_PLANET11:1
INPLAN_TURN_TO_SATELLITE2_STAR7_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_STAR7_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_STAR7_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_STAR7_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_STAR7_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_STAR7_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_STAR7_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_STAR7_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_STAR7_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_STAR7_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_STAR7_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_STAR7_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_STAR7_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_STAR7_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_STAR7_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_STAR6_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_STAR6_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_STAR6_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_STAR6_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_STAR6_PLANET10:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON0_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON4_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR1_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON2_INSTRUMENT5_IMAGE2:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TAKE_IMAGE_SATELLITE2_STAR3_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET11_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET10_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET9_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET8_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR7_INSTRUMENT5_IMAGE0:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_STAR7:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_STAR7:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_STAR7:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_STAR7:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_STAR6:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_STAR6:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_STAR1_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_STAR1_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_STAR1_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_STAR1_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_STAR1_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_STAR1_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_STAR1_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_STAR1_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_STAR1_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_STAR1_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_STAR1_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_STAR1_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_STAR1_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_STAR1_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_STAR1_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_STAR1_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_STAR1_STAR7:0
INPLAN_TURN_TO_SATELLITE2_STAR1_STAR7:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE1_STAR1_STAR7:0
INPLAN_TURN_TO_SATELLITE0_STAR1_STAR7:0
INPLAN_TURN_TO_SATELLITE3_STAR1_STAR6:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET11_INSTRUMENT5_IMAGE1:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_GROUNDSTATAKE_IMAGEON4:1
INPLAN_TURN_TO_SATELLITE2_PLANET11_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_STAR1:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_STAR1:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_STAR1:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_STAR1:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_STAR1:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_STAR1:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_STAR1:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_STAR3:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_STAR3:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_STAR3:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_STAR3:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_STAR6:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_STAR6:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_STAR1:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_STAR1:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_STAR1:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_STAR1:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_STAR3:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_STAR3:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_STAR3:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_STAR3:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_STAR6:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_STAR6:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_STAR1:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_STAR1:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_STAR1:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_STAR1:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_GROUNDSTATAKE_IMAGEON2:1
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_STAR6_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_STAR6_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_STAR6_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_STAR6_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_STAR6_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_STAR6_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_STAR6_PLANET9:1
INPLAN_TURN_TO_SATELLITE3_STAR6_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_STAR6_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_STAR6_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_STAR6_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_STAR6_STAR7:0
INPLAN_TURN_TO_SATELLITE2_STAR6_STAR7:0
INPLAN_TURN_TO_SATELLITE1_STAR6_STAR7:0
INPLAN_TURN_TO_SATELLITE0_STAR6_STAR7:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_PLANET10:1
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_PLANET8:1
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_STAR7:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_STAR7:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_STAR7:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_STAR7:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_STAR6:0
INPLAN_TURN_TO_SATELLITE1_PHENOMENON5_STAR6:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_PLANET11:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_STAR7:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_STAR7:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_STAR7:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_STAR7:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_STAR6:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_STAR6:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON0_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON0_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON0_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON0_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_STAR1_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_STAR1_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_STAR1_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_STAR1_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_STAR1_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_STAR1_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_STAR1_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_STAR1_STAR6:0
INPLAN_TURN_TO_SATELLITE3_STAR1_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_STAR1_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_STAR1_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_STAR1_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_STAR1:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_STAR1:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_STAR1:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_STAR1:0
INPLAN_TURN_TO_SATELLITE3_STAR3_PLANET11:0
INPLAN_TURN_TO_SATELLITE2_STAR3_PLANET11:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE1_STAR3_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_STAR3_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_STAR3_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_STAR3_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_STAR3_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_STAR3_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_STAR3_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_STAR3_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_STAR3_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_STAR3_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_STAR3_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_STAR3_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_STAR3_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_STAR3_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_STAR3_STAR7:0
INPLAN_TURN_TO_SATELLITE2_STAR3_STAR7:0
INPLAN_TURN_TO_SATELLITE1_STAR3_STAR7:0
INPLAN_TURN_TO_SATELLITE0_STAR3_STAR7:0
INPLAN_TURN_TO_SATELLITE3_STAR3_STAR6:0
INPLAN_TURN_TO_SATELLITE1_STAR3_STAR6:0
INPLAN_TURN_TO_SATELLITE3_STAR3_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_STAR3_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_STAR3_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_STAR3_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_STAR3_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_STAR3_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_STAR3_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_STAR3_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_STAR3_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_STAR3_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_STAR3_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_STAR3_STAR1:0
INPLAN_TURN_TO_SATELLITE2_STAR3_STAR1:0
INPLAN_TURN_TO_SATELLITE1_STAR3_STAR1:0
INPLAN_TURN_TO_SATELLITE0_STAR3_STAR1:0
INPLAN_TURN_TO_SATELLITE2_STAR3_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_STAR3_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_STAR3_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET11_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET10_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET9_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET8_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR7_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR6_INSTRUMENT0_IMAGE1:1
INPLAN_TAKE_IMAGE_SATELLITE0_PHENOMENON5_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET10_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET9_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET8_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR7_INSTRUMENT5_IMAGE1:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TAKE_IMAGE_SATELLITE2_STAR6_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PHENOMENON5_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON0_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON4_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR1_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON2_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR3_INSTRUMENT5_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET11_INSTRUMENT6_IMAGE2:1
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET10_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET9_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET8_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR7_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR6_INSTRUMENT6_IMAGE2:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON4_STAR3:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON4_STAR3:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON4_STAR3:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON4_STAR3:0
INPLAN_TURN_TO_SATELLITE2_STAR1_STAR6:1
INPLAN_TURN_TO_SATELLITE0_STAR1_STAR6:1
INPLAN_TURN_TO_SATELLITE1_STAR1_GROUNDSTATAKE_IMAGEON0:1
INPLAN_TURN_TO_SATELLITE3_STAR1_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_STAR1_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_STAR1_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_STAR1_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_STAR1_STAR3:0
INPLAN_TURN_TO_SATELLITE2_STAR1_STAR3:0
INPLAN_TURN_TO_SATELLITE1_STAR1_STAR3:0
INPLAN_TURN_TO_SATELLITE0_STAR1_STAR3:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_STAR6:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_STAR6:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_STAR3:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_STAR3:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_STAR3:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_STAR3:0
INPLAN_TURN_TO_SATELLITE2_STAR3_STAR6:0
INPLAN_TURN_TO_SATELLITE0_STAR3_STAR6:0
INPLAN_TURN_TO_SATELLITE1_STAR3_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_STAR3_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_STAR1:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_STAR1:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_STAR1:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE0_PLANET10_STAR1:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_STAR1:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_STAR1:1
INPLAN_TURN_TO_SATELLITE1_PLANET8_STAR1:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_STAR1:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_STAR3:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_STAR3:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_STAR3:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_STAR3:0
INPLAN_TURN_TO_SATELLITE3_STAR7_STAR6:0
INPLAN_TURN_TO_SATELLITE2_STAR7_STAR6:0
INPLAN_TURN_TO_SATELLITE1_STAR7_STAR6:0
INPLAN_TURN_TO_SATELLITE0_STAR7_STAR6:0
INPLAN_TURN_TO_SATELLITE3_STAR7_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_STAR7_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_STAR7_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_STAR7_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_STAR7_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_STAR7_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE1_STAR7_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_STAR7_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_STAR7_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_STAR7_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_STAR7_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_STAR7_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_STAR7_STAR1:0
INPLAN_TURN_TO_SATELLITE2_STAR7_STAR1:0
INPLAN_TURN_TO_SATELLITE1_STAR7_STAR1:0
INPLAN_TURN_TO_SATELLITE0_STAR7_STAR1:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_PLANET11:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_PLANET11:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_PLANET11:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_PLANET9:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_STAR7:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_STAR7:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_STAR7:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_STAR7:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_STAR6:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_STAR6:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_GROUNDSTATAKE_IMAGEON2_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_GROUNDSTATAKE_IMAGEON2_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_GROUNDSTATAKE_IMAGEON2_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_GROUNDSTATAKE_IMAGEON2_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON0_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON4_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR1_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON2_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR3_INSTRUMENT0_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET11_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET10_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET9_INSTRUMENT0_IMAGE3:1
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET8_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR7_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR6_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_PHENOMENON5_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON0_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON4_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR1_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON2_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR3_INSTRUMENT0_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET11_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET10_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET9_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET8_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR7_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR6_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_PHENOMENON5_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON0_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON4_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR1_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON2_INSTRUMENT1_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR3_INSTRUMENT1_IMAGE3:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TAKE_IMAGE_SATELLITE0_PLANET11_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET10_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET9_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_PLANET8_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR7_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR6_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_PHENOMENON5_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON0_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON4_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR1_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_GROUNDSTATAKE_IMAGEON2_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE0_STAR3_INSTRUMENT2_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_PLANET11_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_PLANET10_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_PLANET9_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_PLANET8_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PHENOMENON5_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON0_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON4_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR1_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON2_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR3_INSTRUMENT6_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET11_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET10_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET9_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET8_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR7_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR6_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PHENOMENON5_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON0_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON4_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR1_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON2_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR3_INSTRUMENT6_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET11_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET10_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET9_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET8_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR7_INSTRUMENT6_IMAGE0:1
INPLAN_TAKE_IMAGE_SATELLITE3_STAR6_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PHENOMENON5_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON0_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON4_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR1_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON2_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR3_INSTRUMENT6_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET11_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET10_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET9_INSTRUMENT7_IMAGE3:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TAKE_IMAGE_SATELLITE3_PLANET8_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR7_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR6_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_PHENOMENON5_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON0_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON4_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR1_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON2_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR3_INSTRUMENT7_IMAGE3:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET11_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET10_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET9_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET8_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR7_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR6_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PHENOMENON5_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON0_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON4_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR1_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON2_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR3_INSTRUMENT7_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET11_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET10_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET9_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PLANET8_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR7_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR6_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_PHENOMENON5_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON0_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON4_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR1_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_GROUNDSTATAKE_IMAGEON2_INSTRUMENT7_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE3_STAR3_INSTRUMENT7_IMAGE1:0
INPLAN_CALIBRATE_SATELLITE0_INSTRUMENT0_STAR1:1
INPLAN_CALIBRATE_SATELLITE0_INSTRUMENT1_GROUNDSTATAKE_IMAGEON0:0
INPLAN_CALIBRATE_SATELLITE0_INSTRUMENT2_GROUNDSTATAKE_IMAGEON2:0
INPLAN_CALIBRATE_SATELLITE1_INSTRUMENT3_GROUNDSTATAKE_IMAGEON4:0
INPLAN_CALIBRATE_SATELLITE2_INSTRUMENT4_STAR1:1
INPLAN_CALIBRATE_SATELLITE2_INSTRUMENT5_STAR1:0
INPLAN_CALIBRATE_SATELLITE3_INSTRUMENT6_GROUNDSTATAKE_IMAGEON4:1
INPLAN_CALIBRATE_SATELLITE3_INSTRUMENT7_GROUNDSTATAKE_IMAGEON0:0
INPLAN_SWITCH_OFF_INSTRUMENT0_SATELLITE0:0
INPLAN_TURN_TO_SATELLITE3_STAR7_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_STAR7_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_STAR7_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_STAR7_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_STAR7_STAR3:0
INPLAN_TURN_TO_SATELLITE2_STAR7_STAR3:0
INPLAN_TURN_TO_SATELLITE1_STAR7_STAR3:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE0_STAR7_STAR3:0
INPLAN_TURN_TO_SATELLITE3_STAR6_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_STAR6_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_STAR6_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_STAR6_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_STAR6_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_STAR6_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE1_STAR6_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_STAR6_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_STAR6_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_STAR3:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_STAR3:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_STAR3:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_STAR3:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_STAR7:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_STAR7:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_STAR7:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_STAR7:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_STAR6:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_STAR6:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_STAR6:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_STAR6:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_STAR1:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_STAR1:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_STAR1:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_STAR1:1
INPLAN_TURN_TO_SATELLITE3_PLANET9_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_PLANET9_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_PLANET9_STAR3:0
INPLAN_TURN_TO_SATELLITE2_PLANET9_STAR3:0
INPLAN_TURN_TO_SATELLITE1_PLANET9_STAR3:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE0_PLANET9_STAR3:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_STAR7:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_STAR7:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_STAR7:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_STAR7:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_STAR6:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_STAR6:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_STAR6:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_STAR6:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_PLANET8_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE1_PLANET8_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_PLANET8_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE3_PLANET8_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE2_STAR6_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TAKE_IMAGE_SATELLITE1_STAR7_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_STAR6_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_PHENOMENON5_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_GROUNDSTATAKE_IMAGEON0_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_GROUNDSTATAKE_IMAGEON4_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_STAR1_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_GROUNDSTATAKE_IMAGEON2_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_STAR3_INSTRUMENT3_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE1_PLANET11_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_PLANET10_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_PLANET9_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_PLANET8_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_STAR7_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_STAR6_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_PHENOMENON5_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_GROUNDSTATAKE_IMAGEON0_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_GROUNDSTATAKE_IMAGEON4_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_STAR1_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_GROUNDSTATAKE_IMAGEON2_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE1_STAR3_INSTRUMENT3_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET11_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET10_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET9_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET8_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR7_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR6_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PHENOMENON5_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON0_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON4_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR1_INSTRUMENT4_IMAGE1:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON2_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR3_INSTRUMENT4_IMAGE1:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET11_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET10_INSTRUMENT4_IMAGE0:1
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET9_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET8_INSTRUMENT4_IMAGE0:1
INPLAN_TAKE_IMAGE_SATELLITE2_STAR7_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR6_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_PHENOMENON5_INSTRUMENT4_IMAGE0:1
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON0_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON4_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR1_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON2_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR3_INSTRUMENT4_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET11_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET10_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET9_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_PLANET8_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR7_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR6_INSTRUMENT5_IMAGE2:0
INPLAN_TAKE_IMAGE_SATELLITE2_PHENOMENON5_INSTRUMENT5_IMAGE2:0
INPLAN_SWITCH_OFF_INSTRUMENT1_SATELLITE0:0
INPLAN_SWITCH_OFF_INSTRUMENT2_SATELLITE0:0
INPLAN_SWITCH_OFF_INSTRUMENT3_SATELLITE1:0
INPLAN_SWITCH_OFF_INSTRUMENT4_SATELLITE2:0
INPLAN_SWITCH_OFF_INSTRUMENT5_SATELLITE2:0
INPLAN_SWITCH_OFF_INSTRUMENT6_SATELLITE3:0
INPLAN_SWITCH_OFF_INSTRUMENT7_SATELLITE3:0
INPLAN_SWITCH_ON_INSTRUMENT0_SATELLITE0:1
INPLAN_SWITCH_ON_INSTRUMENT1_SATELLITE0:0
INPLAN_SWITCH_ON_INSTRUMENT2_SATELLITE0:0
INPLAN_SWITCH_ON_INSTRUMENT3_SATELLITE1:0
INPLAN_SWITCH_ON_INSTRUMENT4_SATELLITE2:1
INPLAN_SWITCH_ON_INSTRUMENT5_SATELLITE2:0
INPLAN_SWITCH_ON_INSTRUMENT6_SATELLITE3:1
INPLAN_SWITCH_ON_INSTRUMENT7_SATELLITE3:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_PLANET10:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_PLANET10:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_PLANET10:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_PLANET10:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_PLANET9:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_STAR7:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10

INPLAN_TURN_TO_SATELLITE2_PLANET11_STAR7:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_STAR7:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_STAR7:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_STAR6:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_STAR6:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_STAR6:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_STAR6:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR6_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_PHENOMENON5_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON0_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON4_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR1_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_GROUNDSTATAKE_IMAGEON2_INSTRUMENT5_IMAGE0:0
INPLAN_TAKE_IMAGE_SATELLITE2_STAR3_INSTRUMENT5_IMAGE0:0
INPLAN_TURN_TO_SATELLITE1_STAR6_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE0_STAR6_GROUNDSTATAKE_IMAGEON4:0
INPLAN_TURN_TO_SATELLITE3_STAR6_STAR1:0
INPLAN_TURN_TO_SATELLITE2_STAR6_STAR1:0
INPLAN_TURN_TO_SATELLITE1_STAR6_STAR1:0
INPLAN_TURN_TO_SATELLITE0_STAR6_STAR1:0
INPLAN_TURN_TO_SATELLITE3_STAR6_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_STAR6_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_STAR6_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_STAR6_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_STAR6_STAR3:0
INPLAN_TURN_TO_SATELLITE2_STAR6_STAR3:0
INPLAN_TURN_TO_SATELLITE1_STAR6_STAR3:0
INPLAN_TURN_TO_SATELLITE0_STAR6_STAR3:0
INPLAN_TURN_TO_SATELLITE2_PHENOMENON5_STAR6:0
INPLAN_TURN_TO_SATELLITE0_PHENOMENON5_STAR6:0
INPLAN_TURN_TO_SATELLITE3_PHENOMENON5_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_STAR1:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_GROUNDSTATAKE_IMAGEON2:0
INPLAN_TURN_TO_SATELLITE3_PLANET11_STAR3:0
INPLAN_TURN_TO_SATELLITE2_PLANET11_STAR3:0
INPLAN_TURN_TO_SATELLITE1_PLANET11_STAR3:0
INPLAN_TURN_TO_SATELLITE0_PLANET11_STAR3:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_PLANET9:0

Solución CSP generada por ClassP utilizando Choco-CP para el problema Satellite mostrado en la Tabla A.10
INPLAN_TURN_TO_SATELLITE2_PLANET10_PLANET9:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_PLANET9:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_PLANET9:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_PLANET8:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_PLANET8:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_PLANET8:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_PLANET8:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_STAR7:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_STAR7:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_STAR7:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_STAR7:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_STAR6:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_STAR6:0
INPLAN_TURN_TO_SATELLITE1_PLANET10_STAR6:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_STAR6:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_PHENOMENON5:1
INPLAN_TURN_TO_SATELLITE1_PLANET10_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE0_PLANET10_PHENOMENON5:0
INPLAN_TURN_TO_SATELLITE3_PLANET10_GROUNDSTATAKE_IMAGEON0:0
INPLAN_TURN_TO_SATELLITE2_PLANET10_GROUNDSTATAKE_IMAGEON0:0

A.6. Dominio ad hoc: MAS-Depots

MAS-Depots es una variación del dominio Depots de la IPC para trabajar con agentes. Consiste en que los almacenes y transportes son codificados como agentes-almacenes y agentes-transportes. Cada almacén tiene un área de carga, un área de almacenamiento y una grúa para mover las cajas en cada una de las paletas de carga.

Tabla A.13: Problema MAS-Depots

Estado Inicial	**Estado Objetivo**
(:init	(:goal
(FICT_AT_E_ORIG)	(ON_A_C_WB)
(FICT_AT_C_ORIG)	(ON_D_A_WB)(ON_B_E_WA)
(FICT_AT_A_ORIG)	

Tabla A.11: Plan solución generado por el planificador LPG-TD para el problema Satellite mostrado en la Tabla A.10

```
; Version LPG-td-1.0
; Seed 123616854
; Problem pfile7
; Time 0.26
; Plan generation time 0.21
; Search time 0.17
; Parsing time 0.02
; Mutex time 0.01
; MakeSpan 99.45

0.0003: (TURN_TO SATELLITE1 STAR1 GS0) [12.9500]
0.0005: (TURN_TO SATELLITE0 STAR1 STAR6) [4.9680]
0.0008: (SWITCH_ON INST0 SATELLITE0) [2.0000]
4.9690: (CALIBRATE SATELLITE0 INST0 STAR1) [10.5000]
4.9693: (TURN_TO SATELLITE0 P9 STAR1) [1.0980]
0.0015: (TURN_TO SATELLITE2 STAR1 STAR6) [4.9680]
15.4698: (TI SATELLITE0 P9 INST0 IMAGE3) [7.0000]
22.4700: (TURN_TO SATELLITE0 STAR6 P9) [23.4600]
0.0022: (SWITCH_ON INST4 SATELLITE2) [2.0000]
4.9705: (CALIBRATE SATELLITE2 INST4 STAR1) [0.3490]
4.9707: (TURN_TO SATELLITE2 P8 STAR1) [14.0100]
18.9810: (TI SATELLITE2 P8 INST4 IMAGE0) [7.0000]
25.9813: (TURN_TO SATELLITE2 PHENOMENON5 P8) [51.9400]
45.9315: (TI SATELLITE0 STAR6 INST0 IMAGE1) [7.0000]
0.0037: (SWITCH_ON INST6 SATELLITE3) [2.0000]
0.0040: (TURN_TO SATELLITE3 GS4 GS2) [41.5000]
41.5042: (CALIBRATE SATELLITE3 INST6 GS4) [24.7000]
41.5045: (TURN_TO SATELLITE3 P11 GS4) [3.9170]
66.2048: (TI SATELLITE3 P11 INST6 IMAGE2) [7.0000]
77.9230: (TI SATELLITE2 PHENOMENON5 INST4 IMAGE0) [7.0000]
73.2053: (TURN_TO SATELLITE3 STAR7 P11) [9.3900]
82.5955: (TI SATELLITE3 STAR7 INST6 IMAGE0) [7.0000]
84.9238: (TURN_TO SATELLITE2 P10 PHENOMENON5) [3.7640]
88.6880: (TI SATELLITE2 P10 INST4 IMAGE0) [7.0000]
95.6882: (TURN_TO SATELLITE2 PHENOMENON5 P10) [3.7640]
```

Tabla A.14: Plan solución generado por el planificador MIPS-XXL para el problema MAS-Depots mostrado en la Tabla A.13

```
; ParsingTime
; NrActions 5
; MakeSpan 3.02
; MetricValue 0.00
; PlaningTechnique ehc
0.00: (AGENTB-AT-LOAD-AREA-E-B ) [1.00]
0.00: (AGENTA-AT-LOAD-AREA-A-A-AT-LOAD-AREA-C-A ) [1.00]
1.01: (AGENTC-AT-LOAD-AREA-C-B-AT-LOAD-AREA-A-B-AT-LOAD-AREA-E-A ) [1.00]
2.02: (AGENTA-ON-B-E-A ) [1.00]
2.02: (AGENTB-ON-A-C-B-ON-D-A-B ) [1.00]
```

Tabla A.15: Solución CSP generada por ClassP utilizando Choco-CP para el problema MAS-Depots mostrado en la Tabla A.13

solve => 1 solutions	
38476844[+0] millis.	
34483856[+0] nodes	
Problem with integer variables	
Num Solutions: 1	S_AGC_AT_LA_E_A:2
	E_AGC_AT_LA_E_A:3
S_AGB_ON_A_C_B_ON_D_A_B_AT_LA_E_B:20	S_AGB_AT_LA_E_B:0
E_AGB_ON_A_C_B_ON_D_A_B_AT_LA_E_B:20	E_AGB_AT_LA_E_B:1
S_AGB_ON_A_C_B_ON_D_A_B:4	S_AGA_AT_LA_A_A_AT_LA_C_A:20
E_AGB_ON_A_C_B_ON_D_A_B:5	E_AGA_AT_LA_A_A_AT_LA_C_A:20
S_AGB_ON_A_C_B_AT_LA_E_B:20	S_AGA_AT_LA_C_A:0
E_AGB_ON_A_C_B_AT_LA_E_B:20	E_AGA_AT_LA_C_A:1
S_AGB_ON_A_C_B:20	S_AGA_AT_LA_A_A:0
E_AGB_ON_A_C_B:20	E_AGA_AT_LA_A_A:1
S_AGB_ON_D_A_B_AT_LA_E_B:20	
E_AGB_ON_D_A_B_AT_LA_E_B:20	INPLAN_START:1
S_AGB_ON_D_A_B:20	INPLAN_END:1
E_AGB_ON_D_A_B:20	INPLAN_AGB_ON_A_C_B_ON_D_A_B_AT_LA_E_B:0
S_AGA_ON_B_E_A_AT_LA_C_A_AT_LA_A_A:20	INPLAN_AGB_ON_A_C_B_ON_D_A_B:1
E_AGA_ON_B_E_A_AT_LA_C_A_AT_LA_A_A:20	INPLAN_AGB_ON_A_C_B_AT_LA_E_B:0
S_AGA_ON_B_E_A_AT_LA_C_A:20	INPLAN_AGB_ON_A_C_B:0
E_AGA_ON_B_E_A_AT_LA_C_A:20	INPLAN_AGB_ON_D_A_B_AT_LA_E_B:0
S_AGA_ON_B_E_A_AT_LA_A_A:20	INPLAN_AGB_ON_D_A_B:0
E_AGA_ON_B_E_A_AT_LA_A_A:20	INPLAN_AGA_ON_B_E_A_AT_LA_C_A_AT_LA_A_A:0
S_AGA_ON_B_E_A:4	INPLAN_AGA_ON_B_E_A_AT_LA_C_A:0
E_AGA_ON_B_E_A:5	INPLAN_AGA_ON_B_E_A_AT_LA_A_A:0
S_AGC_AT_LA_C_B_AT_LA_A_B_AT_LA_E_A:20	INPLAN_AGA_ON_B_E_A:1
E_AGC_AT_LA_C_B_AT_LA_A_B_AT_LA_E_A:20	INPLAN_AGC_AT_LA_C_B_AT_LA_A_B_AT_LA_E_A:0
S_AGC_AT_LA_C_B_AT_LA_A_B:20	INPLAN_AGC_AT_LA_C_B_AT_LA_A_B:0
E_AGC_AT_LA_C_B_AT_LA_A_B:20	INPLAN_AGC_AT_LA_C_B_AT_LA_E_A:0
S_AGC_AT_LA_C_B_AT_LA_E_A:20	INPLAN_AGC_AT_LA_C_W:1
E_AGC_AT_LA_C_B_AT_LA_E_A:20	INPLAN_AGC_AT_LA_E_A_AT_LA_A_B:0
S_AGC_AT_LA_C_W:2	INPLAN_AGC_AT_LA_A_B:1
E_AGC_AT_LA_C_W:3	INPLAN_AGC_AT_LA_E_A:1
S_AGC_AT_LA_E_A_AT_LA_A_B:20	INPLAN_AGB_AT_LA_E_B:1
E_AGC_AT_LA_E_A_AT_LA_A_B:20	INPLAN_AGA_AT_LA_A_A_AT_LA_C_A:0
S_AGC_AT_LA_A_B:2	INPLAN_AGA_AT_LA_C_A:1
E_AGC_AT_LA_A_B:3	INPLAN_AGA_AT_LA_A_A:1
Leyenda	
AG → AGENT	
LA → LOAD_AREA	

Bibliografía

[1] Montserrat Abril. *Particionamiento y resolución distribuida multivariable de problemas de satisfacción de restricciones.* PhD thesis, Universidad Politécnica de Valencia, 2007.

[2] Mitchell Ai-Chang, John Bresina, Len Charest, Adam Chase, Jennifer Cheng-Jung Hsu, Ari Jonsson, Bob Kanefsky, Paul Morris, Kanna Rajan, Jeffrey Yglesias, and et al. MAPGEN: Mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1):8–12, 2004.

[3] Marlene Arangú. Cooperation between planning and scheduling for complex problem resolution. In *11th Ibero-American Conference on AI: Workshop on Planning and Scheduling and Constraint Satisfaction, IBERAMIA-CSPS'08*, pages 19–32, 2008.

[4] Marlene Arangú, Antonio Garrido, and Eva Onaindía. A constraint programming based approach for planning and scheduling problems. In *Problem-Solving In User-Centric Technologies Seminar.*, 2007.

[5] Marlene Arangú, Antonio Garrido, and Eva Onaindía. A general technique for plan repair. In *International Conference on Tools with AI (ICTAI'08)*, volume 1, pages 515–518. IEEE Computer Society Press, 2008.

[6] Marlene Arangú and Miguel Salido. A fine-grained arc-consistency algorithm for non-normalized constraint satisfaction problems. *International Journal of Applied Mathematics and Computer Science, to appear*, 21(4), 2012.

[7] Marlene Arangú, Miguel Salido, and Federico Barber. 2-C3: from arc-consistency to 2-consistency. In *8th Symposium on Abstraction, Reformulation and Approximation, SARA 2009*, pages 184–189, 2009.

[8] Marlene Arangú, Miguel Salido, and Federico Barber. 2-C3OP: an improved version of 2-consistency. In *21st International Conference on Tools with Artificial Intelligence, ICTAI 2009*, pages 344–348, 2009.

[9] Marlene Arangú, Miguel Salido, and Federico Barber. AC3-OP: an arc-consistency algorithm for arithmetic constraints. In S. Sandri et al., editor, *Proceeding of the 2009 conference on Artificial Intelligence Research and Development: Proceedings of the 12th International Conference of the Catalan Association for Artificial Intelligence, CCIA-09*, pages 293–300. IOS Press, 2009.

[10] Marlene Arangú, Miguel Salido, and Federico Barber. AC4-OP: optimizing a fine-grained arc-consistency algorithm. In *XIII Conferencia de la Asociación Española para la Inteligencia Artificial y Jornadas de Transferencia Tecnológica de Inteligencia Artificial, CAEPIA-TTIA 2009*, pages 201–210, 2009.

[11] Marlene Arangú, Miguel Salido, and Federico Barber. Changing propagations in arc-consistency algorithm for arithmetic constraints. In *CAEPIA 2009: Workshop on Planning, Scheduling and Constraint Satisfaction, CSPS-09*, pages 8–20, 2009.

[12] Marlene Arangú, Miguel Salido, and Federico Barber. Extending arc-consistency algorithms for non-normalized CSPs. In *29th SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence, AI-09*, pages 311–316, 2009.

[13] Marlene Arangú, Miguel Salido, and Federico Barber. A filtering technique for non-normalized CSPs. In *15th International Conference on Principles and Practice of Constraint Programming: Doctoral Programme*, pages 1–6, 2009.

[14] Marlene Arangú, Miguel Salido, and Federico Barber. Normalizando CSP no-normalizados: un enfoque híbrido. In *CAEPIA 2009: Workshop on Planning, Scheduling and Constraint Satisfaction, CSPS-09*, pages 57–68, 2009.

[15] Marlene Arangú, Miguel Salido, and Federico Barber. AC2001-OP: An arc-consistency algorithm for constraint satisfaction problems. In *The Twenty Third International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems, IEA-AIE-2010*, volume 3, pages 219–229, 2010.

[16] Marlene Arangú, Miguel Salido, and Federico Barber. A filtering technique for the railway scheduling problem. In *COPLAS 2010: ICAPS 2010 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, pages 68–78, 2010.

[17] Marlene Arangú, Miguel Salido, and Federico Barber. Técnicas de consistencia para problemas de satisfacción de restricciones. In *VI Jornadas de Investigación y Postgrado del Decanato de Ciencias y Tecnología. Universidad Centroccidental Lisandro Alvarado. Barquisimeto, Venezuela*, 2010.

[18] Marlene Arangú, Miguel Salido, and Federico Barber. Artificial intelligence techniques for the railway scheduling problem. *Submitted.*, 2011.

[19] Marlene Arangú, Miguel Salido, and Federico Barber. A filtering technique to achieve 2-consistency in constraint satisfaction problems. *International Journal of Innovative Computing, Information and Control (IJICIC), to appear*, 8(4), 2012.

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

[21] Federico Barber, Montserrat Abril, Miguel Angel Salido, Laura Ingolotti, Pilar Tormos, and Antonio Lova. Survey of automated systems for railway management. Technical report, DSIC-II/01/07. UPV, 2007.

[22] Federico Barber and Miguel Salido. Introduction to constraint programming. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial.*, 20:13–30, 2003.

[23] Roman Barták. Constraint programming: In pursuit of the holy grail. In MatFyzPress, editor, *Proceedings of the Week of Doctoral Students (WDS99), Part IV*, pages 555–564, 1999.

[24] Roman Barták. Theory and practice of constraint propagation. In J. Figwer, editor, *Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control*, 2001.

[25] Roman Barták. Constraint propagation and backtracking-based search. In *Lecture Notes. First international summer school on CP*, 2005.

[26] Roman Barták, Miguel A. Salido, and Francesca Rossi. New trends in constraint satisfaction, planning, and scheduling: a survey. *The Knowledge Engineering Review*, 25(Special Issue 03):249–279, 2010.

[27] Roman Barták and Daniel Toropila. Reformulating constraint models for classical planning. In *Proceedings of the Twenty-First International FLAIRS Conference*, pages 525–530, 2008.

[28] Christian Bessiere. Constraint propagation. Technical report, CNRS, 2006.

[29] Christian Bessiere and Marie Cordier. Arc-consistency and arc-consistency again. In *Proc. of the AAAI'93*, pages 108–113, Washington, USA, July 1993.

[30] Christian Bessiere, E.C. Freuder, and J. C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

[31] Christian Bessiere, Thierry Petit, and Bruno Zanuttini. Making bound consistency as effective as arc consistency. In *Proc. IJCAI 2009*, pages 425–430, 2009.

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

[33] Christian Bessiere, J. C. Régin, Roland Yap, and Yuanling Zhang. An optimal coarse-grained arc-consistency algorithm. *Artificial Intelligence*, 165:165–185, 2005.

[34] Christian Bessiere and Jean-Charles Régin. An arc-consistency algorithm optimal in the number of constraint checks. In *ICTAI'94*, pages 397–403, 1994.

[35] Christian Bessiere and Jean-Charles Régin. Using bidirectionnality to speed-up arc-consistency processing. In *Constraint Processing. Lecture Notes in Computer Science*, pages 157–170. M. Meyer ed., Springer-Verlag, 923, 1995.

[36] Christian Bessiere and Jean-Charles Régin. Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. In *In Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 61–75, 1996.

[37] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Commun. ACM*, 18(11):651–656, 1975.

[38] A.L. Blum and M. L. Furst. Fast planning through planning graph analisys. *Artificial Intelligence*, 90:281–300, 1997.

[39] Assef Chmeiss and Philippe Jegou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7:121–142, 1998.

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

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

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

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

[44] 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.

[45] S. Edelkamp and J. Hoffmann. PDDL2.2: The language for the classical part of 4th international planning competition. In *Technical Report No. 195*, 2004.

[46] Stefan Edelkamp, Shahid Jabar, and Mohammed Nazih. Large-scale optimal pdl-3 planning with MIPS-XXL. In *Proc. Int. Conference on Automated Planning and Scheduling (ICAPS-2006) – International Planning Competition, 2006*, pages 28–30, 2006.

[47] Richard Fikes. Ref-arf: a system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[48] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In *Proc. ICAPS*. AAAI Press, 2006.

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

[50] E. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. *In Proc. of the National Conference on Artificial Intelligence (AAAI-96)*, pages 202–208, 1996.

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

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

[53] A. Garrido and E. Onaindía. *Artificial Intelligence for Advanced Problem Solving. Chapter: Extending classical planning for time: research trends in optimal and suboptimal temporal planning.* Idea Group, Inc., 2007.

[54] Antonio Garrido. *Planificación Temporal Independiente del Dominio. Una aproximación basada en grafos de planificación.* PhD thesis, Universidad Politécnica de Valencia, 2003.

[55] Antonio Garrido, Marlene Arangú, and Eva Onaindía. A constraint programming formulation for planning: from plan scheduling to plan generation. *Journal of Scheduling*, 12:227–256, June 2009.

[56] Antonio Garrido, Maria Fox, and Derek Long. A temporal planning system for durative actions of PDDL2.1. In *ECAI*, pages 586–590. IOS Press, 2002.

[57] Antonio Garrido and Derek Long. Planning with numeric variables multiobjetive planning. In *Proc. 16th European Conference on AI: ECAI-2004*, pages 662–666, 2004.

[58] Antonio Garrido and Eva Onaindía. On the application of planning and scheduling tecniques to e-learning. In *23rd International Conference on Industrial, engineering & Other Application of Applied Intelligent Systems (IEA/AIE-10)*, volume 6096, pages 244–253. Trends in Applied Intelligent Systems. SPRINGER. LNAI, 2010.

[59] Antonio Garrido, Eva Onaindía, and Marlene Arangú. Using constraint programming to model complex plans in an integrated approach for planning and scheduling. In *UK Planning and Scheduling SIG Workshop (PLANSIG)*, pages 137–144, 2006.

[60] Antonio Garrido, Eva Onaindía, and M.G. García-Hernández. Towards an efficient integration of planning and scheduling. In *UK Planning and Scheduling SIG Workshop (PLANSIG)*, pages 58–65, 2005.

[61] Antonio Garrido, Eva Onaindía, and Oscar Sapena. Planning and scheduling in an e-learning environment. a constraint-programming-based approach. *Engineering Applications of Artificial Intelligence*, 21(5):733–743, 2008.

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

[63] 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.

[64] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. the languaje of the fifth international planning competition. In *Technical Report*, 2005.

[65] Alfonso Gerevini, Alessandro Saetti, Ivan Serina, and Paolo Toninelli. LPG-TD: a fully automated planner for PDDL2.2 domains. In *In Proc. of the 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04) International Planning Competition abstracts*, 2004.

[66] Alfonso Gerevini and Ivan Serina. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *AIPS*, pages 112–121, Menlo Park, C.A., 2000. AAAI press.

[67] M. Ghallab and H. Laruelle. Representation and control in ixtet, a temporal planner. In *Proc. Workshop on Planning for Temporal Domains (AIPS-1994)*, pages 61–67, 1994.

[68] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning. Theory and Practice*. Morgan Kaufmann, 2004.

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

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

[71] P. Van Hentenryck, Y. Deville, and C. M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[72] Laura Ingolotti. *Modelos y métodos para la optimización y eficiencia de la programación de horarios ferroviarios.* PhD thesis, Universidad Politécnica de Valencia, 2007.

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

[74] F. Laburthe and N. Jussien. *CHOCO, Choco solver.* http://www.emn.fr/z-info/choco-solver/tex/choco-doc.pdf, 1983.

[75] Javier Larrosa and Pedro Meseguer. Algoritmos para satisfacción de restricciones. *Inteligencia Artificial*, 20, 2003.

[76] Christophe Lecoutre, Frederic Boussemart, and Fred Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *proceedings CP 2003*, pages 480–494, 2003.

[77] Christophe Lecoutre and Fred Hemery. A study of residual supports in arc consistency. In *proceedings IJCAI 2007*, pages 125–130, 2007.

[78] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artif. Intell.*, 173:1592–1614, December 2009.

[79] Christophe Lecoutre and Julien Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2:21–35, 2008.

[80] Chavalit Likitvivatanavong, Yuanlin Zhang, Scott Shannon, James Bowen, and Eugene C. Freuder. Arc consistency during search. In *proceedings IJCAI 2007*, pages 137–142, 2007.

[81] Derek Long and Maria Fox. Language PDDL2.1. In *The International Planning Competition 2002. http://planning.cis.strath.ac.uk/competition/*, 2002.

[82] Derek Long and Maria Fox. The 3rd international planning competition. *Journal of Artificial Intelligence Research, JAIR*, 2003.

[83] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[84] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polinomial network consistency algorithms for constraints satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[85] Felip Manyá and Carla Gomes. Solution techniques for constraint satisfaction problems. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 19:169–180, 2003.

[86] Eliseo Marzal, Eva Onaindía, and Laura Sebastia. Scheduling a plan in time: A csp approach. In *In Proc. of the 15th Int. Conference on Automated Planning and Scheduling (ICAPS-05). Workshop on Constraint Programming for Planning and Scheduling*, pages 44 –51, 2005.

[87] Deepak Mehta. Reducing checks and revisions in the coarse-grained arc consistency algorithms. *CP Letters*, 2:37–53, 2008.

[88] José Tomás Palma Méndez and Roque Marín Morales. *Inteligencia artificial: Métodos, técnicas y aplicaciones*. MIT PressMcGraw-Hill, 2008.

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

[90] Wanlin Pang and Scott D. Goodwin. Binary representations for general csps. In *In Proceedings of 14th Florida AI Research Symposium (FLAIRS-2001), Key West, FL*, 2001.

[91] Federico Pecora and Amedeo Cesta. The role of different solvers in planning and scheduling integration. In *In Proceedings of AI*IA-03*, 2003.

[92] J.S. Penberthy. *Planning with Continuous Change*. PhD thesis, University of Washington, 1993.

[93] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

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

[95] P. Prosser. MAC-CBJ: maintaining arc-consistency with conflict-directed backjumping. *Technical Report 95/177*, 1995.

[96] Jean-Francois Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 359–366, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

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

[98] I. Refanidis. Stratified heuristic pocl temporal planning based on planning graphs and constraint programming. In *In Proc. of the 15th Int. Conference on Automated Planning and Scheduling (ICAPS-05). Workshop on Constraint Programming for Planning and Scheduling*, pages 66 –73, 2005.

[99] J.C. Régin and J.F. Puget. On the equivalence of constraint satisfaction problems. *In Proc. Principles and Practice of Constraint Programming (CP-97)*, pages 32–46, 1997.

[100] Jussi Rintanen. Introduction to automated planning. Technical report, Albert-Ludwigs-Universität Freiburg, 2003-2005.

[101] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier Science y Technology, 2008.

[102] Stuart Russell and Peter Norvig. *Inteligencia Artificial. Un enfoque moderno.* Perason-Prentice Hall, Madrid, España, 2005.

[103] Z. Ruttkay. Constraint Satisfaction - a Survey. *CWI Quarterly*, 11(2&3):123–162, 1998.

[104] 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.

[105] M.A. Salido, F. Barber, M. Abril, L. Ingolotti, A. Lova, P. Tormos, and J. Estrada. Técnicas de inteligencia artificial en planificación ferroviaria. *VI Session on Artificial Intelligence Technology Transfer (TTIA '2005)*, 2005.

[106] Miguel A. Salido. *Polyhedra: Un Modelo para la Resolución de Problemas de Satisfacción de Restricciones N-arias mediante hiper-poliedros.* PhD thesis, Universidad Politécnica de Valencia, 2002.

[107] Miguel A. Salido. A non-binary constraint ordering heuristic for constraint satisfaction problems. *Applied Mathematics and Computation*, 198(1):280–295, 2008.

[108] Miguel A. Salido and Federico Barber. Mathematical solutions for solving periodic railway transportation. *Mathematical Problems in Engineering*, 2009:19, 2009.

[109] Oscar Sapena. *Planificación Independiente del Dominio en Entornos Dinámicos de Tiempo Restringido.* PhD thesis, Universidad Politécnica de Valencia, 2005.

[110] Oscar Sapena and Eva Onaindía. Execution, monitoring and replanning in dynamic environments. In *Workshop on On-Line Planning and Scheduling*, 2002.

[111] Oscar Sapena, Eva Onaindía, Antonio Garrido, and Marlene Arangú. A distributed csp approach for solving multi-agent problems. In *17th International Conference on Automated Planning & Scheduling and 13th International Conference on Principles and Practice of Constraint Programming: Joint Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (CP/ICAPS 2007-COPLAS'07)*, pages 68–75, 2007.

[112] Oscar Sapena, Eva Onaindía, Antonio Garrido, and Marlene Arangú. A distributed csp approach for collaborative planning systems. *Engineering Applications of Artificial Intelligence*, 21(5):698–709, 2008.

[113] Sun Developer Network SDN. *Java Technology.* http://java.sun.com/, 2008.

[114] Elias. Silva de Oliveira. *Solving Single-Track Railway Scheduling Problem Using Constraint Programming.* PhD thesis, University of Leeds, School of Computing, 2001.

[115] David E. Smith, Jeremy Frank, and Ari K. Jónsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15:1–34, 2000.

[116] P. Tormos, A. Lova, F. Barber, L. Ingolotti, M. Abril, and M.A. Salido. A genetic algorithm for railway scheduling problems. *Metaheuristics for Scheduling In Industrial and Manufacturing Applications, ISBN: 978-3-540-78984-0*, pages 255–276, 2008.

[117] Ioannis Tsamardinos and Martha E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1-2):43–89, 2003.

[118] Roman van der Krogt and Mathijs de Weerdt. Plan repair as an extension of planning. In *ICAPS*, pages 161–170, 2005.

[119] Roman van der Krogt and Mathijs dee Weerdt. The two faces of plan repair. In *BNAIC*, 2004.

[120] M.R.C. van Dongen, A.B. Dieker, and A. Sapozhnikov. The expected value and the variance of the checks required by revision algorithms. *CP Letters*, 2:55–77, 2008.

[121] Hector Vidal, Vincent y Geffner. Branching and prunning: An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence*, 170:298–335, 2006.

[122] Cameron Walker, Jody Snowdon, and David Ryan. Simultaneous disruption recovery of a train timetable and crew roster in real time. *Comput. Oper. Res*, pages 2077–2094, 2005.

[123] D.L. Waltz. *Generating Semantic description from drawings of scenes with shadows.* Technical Report AI-TR-271, MIT, Cambridge, 1972.