PARTICIONAMIENTO Y RESOLUCIÓN DISTRIBUIDA MULTIVARIABLE DE PROBLEMAS DE SATISFACCIÓN DE RESTRICCIONES

Autor: Montserrat Abril López

Directores: Dr. D. Federico Barber Sanchis

Dr. D. Miguel Ángel Salido Gregorio

PARA LA OBTENCIÓN DEL GRADO DE
DOCTOR EN INFORMÁTICA
POR LA
UNIVERSIDAD POLITÉCNICA DE VALENCIA

Valencia, España

DICIEMBRE 2007

Fecha: **Diciembre 2007**

Autor: **Montserrat Abril López**

Directores: **Dr. D. Federico Barber Sanchis**

Dr. D. Miguel Ángel Salido Gregorio

Título: **Particionamiento y resolución distribuida multivariable de problemas de satisfacción de restricciones**

Departamento: **Sistemas Informáticos y Computación**

Universidad: **Universidad Politécnica de Valencia**

Grado: **Doctor** Mes: **Diciembre** Año: **2007**

Firma del Autor

A mis padres.

Tabla de Contenidos

Índice de figuras

Agradecimientos

El primer y principal agradecimiento va dirigido a mis directores de tesis, Federico Barber y Miguel Ángel Salido. Ellos me introdujeron en el programa de doctorado y han trabajado junto a mí para sacar adelante esta tesis. Sin su habilidad y paciencia para organizar mis confusiones y su confianza continua en este trabajo, no creo que hubiese sido capaz de llegar a buen término.

A los miembros del Grupo de *Inteligencia Artificial, Planificación, Scheduling y Razonamiento Basado en Restricciones*: Antonio, Federico, Laura, Miguel Ángel y Pilar, les dedico un agradecimiento muy especial. Su amistad, su ayuda en la investigación y los buenos ratos pasados juntos no tienen precio.

También quería expresar mi agradecimiento a los miembros de la *Unidad de Circulación* de *ADIF*, particularmente al director de Gestión de Capacidades, José Estrada, sus conocimientos y experiencias en el campo ferroviario me han sido de gran ayuda para incorporar una visión práctica en el trabajo realizado.

La gente del *DSIC* se merece muchas y buenas palabras, ya que siempre han colaborado conmigo brindándome una sonrisa. Mención aparte merecen mis amigas y compañeras de despacho, Laura, Mariamar y Marlene, y otros muy queridos compañeros como Adriana, Antonio y Oscar, entre todos hemos conseguido crear un ambiente de trabajo muy difícil de superar, lo cual ha contribuido enormemente a que todo esto fuese más llevadero.

Por supuesto, quiero agradecer a mis familiares y amigos. Muy especialmente a mis padres, Constantino y Mª Rosa, por darme la oportunidad de recibir una formación a la carta, y que junto a mis hermanos, Raúl y Lidia, me han brindado un apoyo incondicional a lo largo de todos estos años.

Finalmente, a todas aquellas personas que de una u otra forma, colaboraron en la realización de esta tesis, hago extensivo mi más sincero agradecimiento.

Montserrat Abril López

Resumen

Hoy en día, muchos problemas reales pueden modelarse como problemas de satisfacción de restricciones (CSPs) y se resuelven usando técnicas específicas de satisfacción de restricciones. Estos problemas pertenecen a áreas tales como Inteligencia Artificial, investigación operativa, sistemas de información, bases de datos, etc. La mayoría de estos problemas pueden modelarse de forma natural mediante CSPs. Sin embargo, algunos de estos problemas son de naturaleza distribuida, bien por motivos de seguridad, por requerimientos de privacidad, o por restricciones espaciales. Ello requiere que este tipo de problemas sean modelados como problemas de satisfacción de restricciones distribuidos (DCSPs), donde el conjunto de variables y restricciones del problema está distribuido entre un conjunto de agentes que se encargan de resolver su propio sub-problema y deben coordinarse con el resto de agentes para alcanzar una solución al problema global.

En la literatura de satisfacción de restricciones, la necesidad de manejar DCSP comenzó a tratarse a principios de los 90. No obstante, la mayoría de los investigadores que trabajan en este campo centran su atención en algoritmos en los que cada agente maneja una única variable. Estos algoritmos pueden ser transformados para que cada agente maneje múltiples variables. Sin embargo, los algoritmos resultantes no son escalables para manejar grandes sub-problemas locales debido tanto a requerimientos espaciales como a su coste computacional. Por lo tanto, la resolución de problemas reales mediante este tipo de algoritmos resulta prácticamente inviable.

En esta tesis presentamos nuevos algoritmos para la resolución de problemas de satisfacción de restricciones distribuidos capaces de manejar multiples variables por agente. Estos algoritmos realizan un manejo eficiente de la información obtenida mediante la comunicación entre los agentes, consiguiendo con ello una mayor eficiencia durante el proceso de resolución. Además, sus requerimientos espaciales son mínimos, por lo que son capaces de manejar grandes sub-problemas locales. También se presentan en esta tesis varios algoritmos y técnicas heurísticas para que cada agente realice eficazmente la búsqueda de soluciones parciales de su sub-problema.

Por otra parte, los problemas de satisfacción de restricciones suelen ser aplicados sobre grandes problemas reales, que generalmente implican modelos con un gran número de variables y restricciones y en los cuales, a menudo, se suelen identificar sub-problemas más reducidos. Esta clase de problemas puede ser manejada de manera global sólo a costa de un exagerado coste computacional. Esto motiva una contribución adicional de esta tesis que se basa en proporcionar unas metodologías para particionar problemas de gran volumen o con entidades diferenciales, para posteriormente poder ser resueltos más eficazmente mediante los algoritmos propuestos para resolver DCSPs. En esta tesis planteamos tres modos para particionar un problema: el primero se basa en técnicas de particionamiento de grafos, el segundo en la identificación de estructuras de árbol y el tercero en la identificación de entidades propias de cada problema. Tanto las propuestas de particionamiento de CSPs, como los nuevos algoritmos de resolución de DCSPs, constituyen el núcleo de las aportaciones de esta tesis. Finalmente se evalúan los métodos propuestos sobre escenarios de prueba, se efectúa una revisión crítica y se plantean líneas de futuro.

Abstract

Nowadays, many real problems can be modelled as constraint satisfaction problems (CSPs) and solved using specific constraint satisfaction techniques. These include problems from fields such as artificial intelligence, operational research, information systems, databases, etc. Most of these problems can be naturally modelled using CSPs. However, some of these problems are of a distributed nature either for reasons of security, for privacy requirements, or for spatial constraints; this requires that problems of this kind must be modelled as distributed constraint satisfaction problems (DCSPs), where the set of variables and constraints of the problem are distributed among a set of agents who are in charge of solving their own sub-problem and must coordinate themselves with the rest of the agents in order to reach a solution to the global problem.

In the Literature of constraint satisfaction, the need to handle DCSP emerged at the beginning of the 90s. However, most researchers who work in this field focus their attention on algorithms in which each agent handles a single variable. Even though these algorithms can be transformed so that each agent handles multiple variables, none of the resulting algorithms scales up well as the size of the problems increases due to space requirements and/or computacional cost. Therefore, the resolution of real problems with algorithms of this type, in practice, is not viable.

In this thesis, we present new algorithms for the resolution of distributed constraint satisfaction problems that are able to handle multiple variables by

agent. These algorithms handle the data obtained by means of communication between the agents in order to obtain greater efficiency during the resolution process. In addition, their space requirements are minimum which is why they are able to handle large, local sub-problems. Several algorithms and heuristic techniques are presented in this thesis so that each agent carries out effectively the search of partial solutions of his sub-problem.

In addition, constraint satisfaction problems can also model large real problems, they generally imply models with a great number of variables and constraints.These problems often have clear, smaller sub-problems. These problems can be handled as a whole only at overwhelming computational cost. This leads to one more contribution of this thesis: it is based on the problem partition so that, they can subsequently be solved more effectively by means of the proposed algorithms that solve DCSPs. In this thesis we propose three ways to divide a problem: first one is based on techniques of graph partitioning, the second is based on the identification of tree structures and third is based on the identification of problem common entities. Both the partitioning methodologies and the algorithms to solve DCSPs represent the heart of this thesis. Finally, we evaluate the proposed method, make a review and think about the future work.

Resùm

Hui en dia, molts problemes reals poden modelar-se com problemes de satisfacció de restriccions (CSPs) i es resolen usant tècniques específiques de satisfacció de restriccions. Estos problemes pertanyen a àrees com ara la Intel·ligència Artificial, la Investigació Operativa, sistemes d'informació, bases de dades, etc. La majoria d'estos problemes poden modelar-se de forma natural per mitjà de CSPs. No obstant això, alguns d'estos problemes són de naturalesa distribuïda, bé per motius de seguretat o bé per requeriments de privacitat o restriccions espacials; açò requereix que aquest tipus de problemes es modelitzen com a problemes de satisfacció de restriccions distribuïts (DCSPs), on el conjunt de variables i restriccions del problema està distribuït entre un conjunt d'agents que s'encarreguen de resoldre el seu propi subproblema i han de coordinar-se amb la resta d'agents per a aconseguir una solució al problema global.

En la literatura de satisfacció de restriccions, la necessitat de manejar DCSP va començar a tractar-se a principis dels 90. No obstant això, la majoria dels investigadors que treballen en aquest camp centren la seua atenció en algorismes on cada agent maneja una única variable. Estos algorismes poden ser transformats perquè cada agent manege múltiples variables, encara que els algorismes resultants no són escalables per a manejar grans subproblemes locals degut tant als requeriments espacials com al seu cost computacional. Per tant, la resolució de problemes reals per mitjà d'aquest tipus d'algorismes resulta pràcticament inviable.

En esta tesi presentem nous algorismes per a la resolució de problemes de satisfacció de restriccions distribuïts capaços de manejar múltiples variables per agent. Estos algorismes realitzen un maneig eficient de la informació obtinguda per mitjà de la comunicació entre els agents per a aconseguir una major eficiència durant el procés de resolució. A més, els seus requeriments espacials són mínims, per la qual cosa és capaç de manejar grans subproblemes locals. Diversos algorismes i tècniques heurístiques són presentades en esta tesi perquè cada agent realitze eficaçment la cerca de solucions parcials del seu subproblema.

A més, els problemes de satisfacció de restriccions poden també modelar grans problemes reals, que generalment impliquen models amb un gran nombre de variables i restriccions. Aquest tipus de problemes pot ser manejat de manera global únicament mitjançant un exagerat cost computacional. Açò motiva la següent contribució d'esta tesi que consisteix en particionar aquest tipus de problemes per a tractar-los posteriorment més eficaçment per mitjà dels algorismes proposats per a resoldre DCSPs. En esta tesi plantegem tres formes de particionar un problema: la primera es basa en tècniques de particionament de grafs, la segona en la identificació d'estructures d'arbre i la tercera en la identificació d'entitats pròpies de cada subproblema. Tant les propostes de particionamiento de CSPs, com els nous algoritmes de resolució de DCSPs, constituïxen el nucli de les aportacions d'esta tesi. Finalment s'avaluen els mètodes proposats sobre escenaris de prova, s'efectua una revisió crítica i es plantegen línies de futur.

Capítulo 1

Introducción

En el mundo real existen numerosas áreas tales como planificación, scheduling, diagnosis, configuración, diseño, logística, etc., cuya tipología de problemas puede modelarse como un problema de satisfacción de restricciones (CSP, Constraint Satisfaction Problem). Este tipo de problemas ha formado parte de las inquietudes de la comunidad científica desde hace mucho tiempo. Tradicionalmente los CSPs han sido resueltos de manera centralizada, esto es claramente una limitación de su utilidad para la resolución de muchos problemas reales, ya sea por la naturaleza distribuida del problema o por la dimensión del mismo. De aquí surgió la necesidad de nuevas técnicas para la resolución de problemas de satisfacción de restricciones distribuidos (DCSP, Distributed Constraint Satisfaction Problem) los cuales nos permitan la resolución de un CSP de una manera distribuida. Esta es la motivación principal de esta tesis doctoral que consiste en el desarrollo de técnicas distribuidas para la resolución de CSPs. A continuación realizamos una presentación de los problemas de satisfacción de restricciones, tanto centralizados como distribuidos, así como las motivaciones que nos han conducido al desarrollo de la presente tesis y las contribuciones que con ella aportamos.

1.1. Generalidades

La programación por restricciones es una metodología software utilizada para la descripción y posterior resolución efectiva de cierto tipo de problemas, típicamente combinatorios y de optimización. Estos problemas aparecen en muy diversas áreas, incluyendo inteligencia artificial, investigación operativa, bases de datos y sistemas de recuperación de la información, etc. Algunos ejemplos son scheduling, planificación, razonamiento temporal, diseño en la ingeniería, problemas de empaquetamiento, criptografía, diagnóstico, toma de decisiones, etc. Estos problemas pueden modelarse como problemas de satisfacción de restricciones (CSPs) y resolverse usando técnicas de satisfacción de restricciones. En general, se trata de grandes y complejos problemas, típicamente de complejidad NP-completo.

Las etapas básicas para la resolución de un problema CSP son su modelización y su posterior resolución mediante la aplicación de técnicas CSP específicas, que incluyen procesos de búsqueda apoyados con métodos heurísticos y procesos inferenciales.

Los primeros trabajos relacionados con la programación de restricciones datan de los años 60 y 70 en el campo de la Inteligencia Artificial. Durante los últimos años, la programación de restricciones ha generado una creciente expectación entre expertos de muchas áreas debido a su potencial para la resolución de grandes y complejos problemas reales. Sin embargo, al mismo tiempo, se considera como una de las tecnologías menos conocida y comprendida. La programación de restricciones se define como el estudio de sistemas computacionales basados en restricciones. La idea de la programación de restricciones es resolver problemas mediante la declaración de restricciones sobre el área del problema y consecuentemente encontrar soluciones que satisfagan todas las restricciones y, en algunos casos, optimicen unos criterios determinados.

"*Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the*

problem, the computer solves it"(E. Freuder).

El interés actual por el desarrollo e investigación en técnicas de satisfacción de restricciones está suficientemente contrastado en los diferentes foros científicos, bibliografía relevante, aplicaciones destacadas, etc.

Las dos principales cuestiones con las que nos encontramos a la hora de resolver un problema de satisfacción de restricciones son, primero, cómo representar el problema, y segundo cómo resolverlo. En esta tesis investigamos y comparamos las diferentes maneras de resolver problemas de satisfacción de restricciones.

La programación de restricciones puede dividirse en dos ramas claramente diferenciadas: la "*satisfacción de restricciones"* y la "*resolución de restricciones*". Ambas comparten la misma terminología, pero sus orígenes y técnicas de resolución son diferentes. La satisfacción de restricciones trata con problemas (CSP) que tienen dominios finitos, mientras que la resolución de restricciones está orientada principalmente a problemas sobre dominios infinitos o dominios más complejos.

En esta tesis se tratarán principalmente los problemas de satisfacción de restricciones. Los conceptos clave en la metodología CSP corresponden a los aspectos de:

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

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

- Técnicas de búsqueda de la solución, apoyadas generalmente por criterios

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

Respecto al paradigma distribuido, los problemas de satisfacción de restricciones distribuidas surgen cuando es importante independizar cierta información relativa a variables, restricciones del problema, o ambos por medio de agentes de comunicación [117]. Esto proporciona un entorno prometedor para tratar una amplia variedad de problemas reales que por naturaleza son inherentemente distribuidos. Además, muchos problemas típicos en sistemas multiagente, que tratan de encontrar una combinación consistente entre las acciones de los agentes (problemas de asignación de recursos distribuidos, problemas de scheduling distribuido, sistemas de mantenimiento de verdad), pueden ser formalizados como CSPs distribuidos. Los nuevos desafíos propuestos por la comunidad científica que trabaja con los CSPs distribuidos incluyen manejar restricciones sobre los recursos, explotar las oportunidades para la cooperación, así como diseñar estrategias para la resolución de problemas complejos. Por otra parte, uno de los desafíos más importantes en la mayoría de los problemas reales de hoy en día, son los requerimientos de privacidad [103]. En una sociedad abierta, donde los problemas reales abarcan intereses en diferentes sectores de la sociedad, es necesaria una medida de robustez que mantenga la privacidad de la información aportada al problema [120]. De esta manera, el objetivo de los CSPs distribuidos es doble, por un lado obtener una solución de la manera más eficiente posible, y por otro, la consistencia de dicha solución frente a variaciones del problema, teniendo en cuenta la privacidad de la información. Adicionalmente, la aplicación de CSPs distribuidos está teniendo un interés reciente en su aplicación a problemas de escalamiento incremental

en complejidad, dinamicidad, aleatoriedad, e incluso en el caso de que existan situaciones adversas en las que el fallo en la resolución de una parte del problema pueda resultar muy costoso o catastrófico [1]. En relación a estos métodos y sus fronteras abiertas, resulta de interés desarrollarlos para la obtención de soluciones robustas en complejos problemas, así como diseñar y desarrollar técnicas apropiadas para obtener soluciones estables y robustas en CSP Distribuidos.

Precisamente, el advenimiento de la computación distribuida y ubicua, redes de telecomunicaciones y sensores, proporcionan un nuevo entorno en el que es posible formular problemas más complejos y resolverlos por la propia integración de sistemas individuales. De esta manera, es posible hibridizar las técnicas previas en los llamados sistemas multi-agente. Existen diferentes trabajos sobre la aplicación de sistemas multi-agente para la resolución de los problemas de scheduling [70], [65] formulados como problemas de satisfacción de restricciones, entre los que se pueden mencionar: backtracking asíncrono [22], búsqueda asíncrona con compromisos débiles [117], y algoritmos distribuidos de mejoras iterativas [119]. Todos los métodos presentan una serie de limitaciones, entre las que se encuentra el hecho de heredar algunas características de los métodos no distribuidos. Desde un punto de vista multi-agente, se pueden enfocar otras soluciones a un nivel de abstracción más elevado, esto es, mediante la programación orientada al agente [34]. Además, se pueden adaptar otros paradigmas como los provenientes de las teorías económicas (subastas combinatorias [53], [14])y ecosistemas ([52], [82]) que proporcionan enfoques alternativos y novedosos. Sin embargo la utilización de estas técnicas provenientes de otras disciplinas requiere de revisiones importantes que tengan en cuenta aspectos como la importancia de la ordenación en la secuenciación, la prevención para gestionar incidencias, y la dinámica propia de problemas complejos.

Otro tópico de interés en esta tesis es el particionamiento del CSP. Cuando hablamos del particionamiento de un CSP lo que se pretende es dividir el conjunto de variables en varios sub-conjuntos de manera que cada sub-conjunto

de variables junto a las restricciones que relacionan esas variables constituya un sub-CSP. Este problema está muy relacionado con el problema del particionamiento de grafos ya que un CSP binario puede representarse como un grafo donde cada nodo representa una variable y cada arista una restricción. Generalmente cuando se habla de particionamiento de grafos lo que se pretende es conseguir dividir el grafo en sub-conjuntos de nodos; de manera que cada sub-conjunto tenga aproximadamente el mismo número de nodos y haya el mínimo número de aristas conectando los nodos de diferentes sub-conjuntos. Sin embargo, hay otros tipos de particionamiento [78] que lo que pretenden es conseguir una estructura particular formada por los nodos de cada sub-conjunto.

1.2. Problema de satisfacción de restricciones

Un problema de satisfacción de restricciones (CSP) consiste en un conjunto finito de variables, un dominio de valores para cada variable y un conjunto de restricciones que acotan la combinación de valores que las variables pueden tomar.

El objetivo de un CSP es seleccionar un valor para cada variable de manera que se satisfagan todas las restricciones del problema.

Los CSPs son, en general, problemas intratables, es decir, no se conocen algoritmos polinómicos para resolver tales problemas. Por ejemplo un problema con 10 variables, y cada variable con 10 posibles valores, tendría un total de diez mil millones de posibilidades diferentes. Los algoritmos completos para llevar a cabo la búsqueda de soluciones en un CSP están basados en técnicas hacia atrás (backtracking). Consideramos algoritmos completos a aquellos que garantizan encontrar una solución si existe, o probar la insatisfactibilidad en caso contrario. Los algoritmos de backtracking asignan valores a variables, uno a uno, hasta que se encuentre una asignación consistente de todas las variables, o hasta que se hayan probado sin éxito todas las combinaciones posibles. Cada

vez que se le asigna un valor a una variable, los algoritmos de backtracking comprueban si ese valor es compatible con todos los asignados previamente con respecto a las restricciones del problema. Si se cumple esta condición el algoritmo asigna un valor a la siguiente variable. En caso contrario, la ultima asignación realizada no es válida, y se le asigna el siguiente valor del dominio. En el caso de que se hayan probado todos los valores posibles de la variable, y hayan fallado, el algoritmo retrocede a la variable asignada anteriormente y asigna el siguiente valor de su dominio. De esta manera si llamamos n al número de variables y d a la talla del dominio de todas las variables, entonces el número de posibles combinaciones es d^n. Esto supone una complejidad de $O(d^n)$.

1.2.1. Ejemplo de CSP

El problema de coloración del mapa puede ser formulado como un CSP. En este problema, hay que colorear cada región de un mapa mediante un conjunto de colores de manera que las regiones adyacentes tengan distintos colores. En la formulación del CSP, hay una variable por cada región del mapa, y el dominio de cada variable es el conjunto de colores disponible. Para cada par de regiones contiguas existe una restricción sobre las variables correspondientes que no permite la asignación de idénticos valores a las variables. Dicho mapa puede ser representado mediante un grafo donde los nodos son las regiones y cada par de regiones adyacentes están unidas por una arista. En la Figura 1.1 se muestra el ejemplo del problema de coloración del mapa. El problema consta de cuatro regiones $\{x, y, z, w\}$ para ser coloreadas. Cada región del mapa se corresponde con una variable en el grafo. Si asumimos que cada región puede colorearse con uno de los tres colores, rojo (r), verde (v) y azul (a), entonces cada variable del grafo tiene tres posibles valores.

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

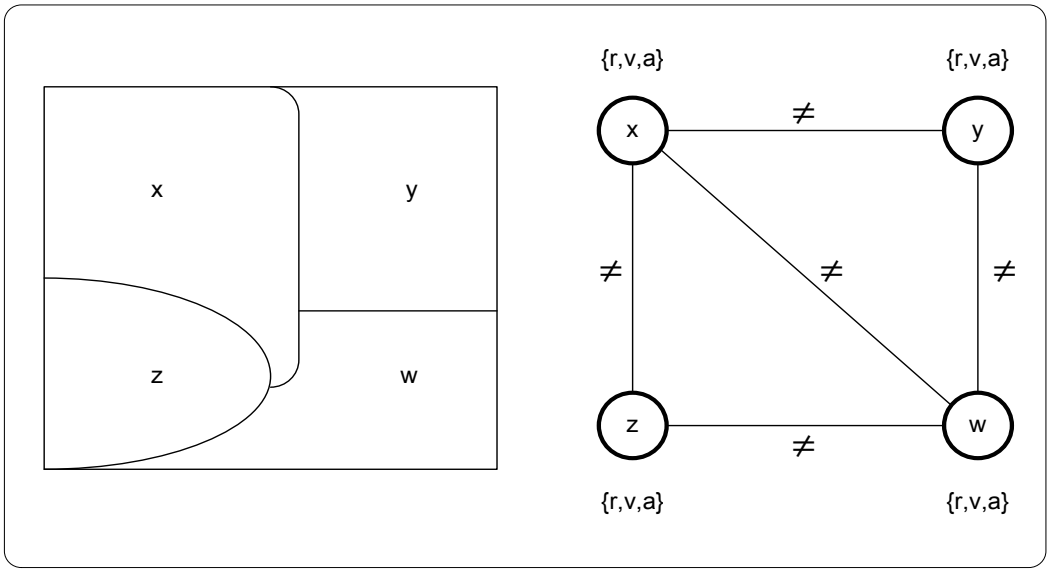


Figura 1.1: Problema de coloración del mapa.

arista. Hay cinco restricciones en el problema, es decir, cinco aristas en el grafo. Una solución para el problema es la asignación (x, r), (y, v), (z, v), (w, a). En esta asignación todas las variables adyacentes tienen valores diferentes.

1.3. Problema de satisfacción de restricciones distribuido

Un problema de satisfacción de restricciones distribuido (DCSP) se formaliza como un CSP donde las variables y restricciones son distribuidas entre multiples agentes automatizados.

Los problemas de satisfacción de restricciones distribuidos son todavía una disciplina joven. Las primeras publicaciones aparecieron a finales de los 80 y principios de los 90, y la mayor parte del trabajo se realizó a mediados de los 90, después de los pioneros trabajos de Yokoo y sus colegas [116][115][118]. A mediados de los 90, aparecieron muchas otras publicaciones y finalmente se convirtió en un tópico popular.

Las diferentes aproximaciones distribuidas planteadas para resolver DCSP pueden clasificarse en función de varios criterios:

- Si el control de la búsqueda es centralizado o distribuido;
- si el protocolo de comunicación tiene la memoria compartida o se realiza mediante el paso de mensajes;
- si los requerimientos de memoria son polinomiales o no;
- si el algoritmo es completo o no;
- si la ordenación de agentes o variables es estática o dinámica;
- si la distribución del problema se basa en variables o restricciones.

En la última década, se han hecho grandes progresos en la satisfacción de restricciones distribuidas . Se han desarrollado varios algoritmos para realizar búsqueda completa o local distribuida. Entre los algoritmos que realizan búsqueda completa están los algoritmos asíncronos [119], con esquemas de ordenación de variables distribuida [46] y con forward checking distribuido [77] [74] [83]. Entre los algoritmos de búsqueda local distribuida están el algoritmo breakout distribuido [118], el algoritmo de búsqueda estocástica distribuida [121] y el algoritmo de satisfacción de restricciones paralelo distribuido [37].

Aunque de manera genérica todos estos algoritmos tienen la posibilidad de manejar varias variables por agente, son muy pocos los trabajos que exploran las peculiaridades del manejo de sub-problemas locales complejos: la complejidad de la instanciación del agente, lo cual, en este caso, no puede ser visto como una operación casi instantánea; el incremento en la necesidad de colaboración entre los agentes; y la generación de mensajes con valores inconsistentes (Nogood) de manera inteligente. Existen algunos trabajos relacionados con la ordenación de las variables y/o agentes, lo cual es típicamente más importante cuando aumenta la granularidad de los agentes locales, ya que una buena ordenación suele conducir a un mejor rendimiento.

Además del desarrollo de algoritmos, también se han investigado otros aspectos en el marco de los DCSP. A continuación se resumen algunos de ellos:

- **Resolución de problemas de gran escala.** Uno de los mayores desafíos para la comunidad investigadora de DCSP es el desarrollo de algoritmos capaces de manejar problemas de gran escala. Hasta ahora, los algoritmos distribuidos de búsqueda sistemática tienen grandes dificultades para resolver en un tiempo aceptable problemas que involucren más de 30 variables. Por esto, ninguna aplicación a problemas reales ha sido realizada usando un algoritmo para DCSP. Uno de los mayores problemas en los algoritmos DCSP completos es el gran número de mensajes que se intercambian. Sólo los algoritmos de búsqueda local distribuidos son capaces de resolver problemas de gran tamaño. Sin embargo, el inconveniente de estos últimos es que son incompletos: pueden perder soluciones y entrar en un ciclo. Un gran desafío es construir un algoritmo que resuelva problemas de satisfacción de restricciones distribuido pero que lleve a cabo un menor número de intercambios de mensajes.

- **Protocolos de ejecución paralela.** Los problemas de DCSP son resueltos por varios procesos. Cuando se usan protocolos de ejecución síncrona, suelen producirse espacios de tiempo en los que el procesador está inactivo, por ejemplo cuando un agente está esperando un mensaje. Este tiempo de inactividad podría ser utilizado para resolver problemas en paralelo. Los protocolos asíncronos son un paso en esta dirección. Otros métodos, dividen el espacio de búsqueda en subproblemas y los resuelven independientemente. Otra opción es construir algoritmos que resuelvan los DCSP optimizando el uso de toda la capacidad disponible en el procesador. Se han realizado algunos trabajos en ejecución paralela, tales como distributed parallel search [123], o algoritmos de búsqueda cooperativa, paralela, intercalada y distribuida [45].

- **Problemas de contexto dinámico.** Muchas aplicaciones prácticas de

DCSP están enmarcadas en un contexto dinámico, donde las variables, los valores y las restricciones son añadidas, borradas o modificadas continuamente. Para que los algoritmos de DCSP puedan tratar con estos problemas, deben ser flexibles, robustos y permitir el cambio de la información incluso durante el proceso de búsqueda. Relativamente pocos trabajos [47] se han realizado para contextos dinámicos. La mayoría se focalizan en contextos estáticos [117].

- **Factores perjudiciales.** Cuando una aplicación está distribuida sobre una red, la probabilidad de fallo incrementa. Algunos nodos pueden no estar disponibles, pueden producirse retrasos, los mensajes pueden llegar no sincronizados: en un orden distinto al que fueron enviados, e incluso pueden llegar a perderse. Para cubrir estos asuntos, los algoritmos DCSP deberían ser flexibles y robustos. La resolución asíncrona es una manera de aumentar la flexibilidad de los algoritmos. Cuando un agente no responde o se pierden sus mensajes, el resto de agentes pueden continuar con la resolución. En algoritmos síncronos, esto no es posible, ya que cuando un mensaje se pierde, el algoritmo tiene que parar. Otra posibilidad es escoger a priori un algoritmo robusto. Los algoritmos de búsqueda local en general son robustos y parecen ser una mejor elección cuando se comparan con los algoritmos de búsqueda completa. Algoritmos considerados robustos de búsqueda local son por ejemplo, el algoritmo distribuido breakout [118], el algoritmo de búsqueda estocástica distribuido [121] [38], y el algoritmo de satisfacción de restricciones paralelo [37].

- **Requerimientos de privacidad.** Uno de los principales motivos para desarrollar algoritmos para DCSP, es mantener la información distribuida y privada. Sin embargo, durante la búsqueda, siempre se revela alguna información privada mediante la asignación de valores consistentes o inconsistentes, estos últimos se suelen llamar *Nogoods*. A medida que la búsqueda se hace más prolongada, las variables y restricciones se

suelen ir haciendo más trasparentes. Por otro lado, los agentes nunca pueden saber si la información que tienen del dominio de otras variables está completa, ya que el dominio puede tener más valores, los cuales nunca se seleccionaron porque violaban alguna restricción. Además las restricciones nunca son directamente reveladas y un *Nogood* contiene la información sumamente resumida como resultado de muchas restricciones.

Los algoritmos distribuidos de búsqueda local tienen mejores propiedades de privacidad que los algoritmos distribuidos de búsqueda completa. Los algoritmos distribuidos de búsqueda completa exploran el espacio de búsqueda sistemáticamente, esto facilita la deducción de información privada. Los algoritmos distribuidos de búsqueda local exploran el espacio de búsqueda de manera irregular y por esto la deducción de información privada es más difícil.

El reto actual es desarrollar algoritmos para DCSP que tengan mejores propiedades de privacidad. Esto no solo desde el punto de vista de la deducción de información privada, sino también para la seguridad de los datos. Yokoo y otros [120] han desarrollado un algoritmo seguro para DCSP que usa un método de encriptación con clave pública para proteger los datos. En este algoritmo el problema se resuelve sobre un conjunto de servidores, los cuales reciben la información encriptada desde los agentes. Este algoritmo es totalmente seguro y no desvela ninguna información privada. Los autores muestran que ni los agentes ni los servidores pueden obtener información adicional de la asignación de valores de las variables de otros agentes. Existen otros trabajos [104], [77] que también tratan con requerimientos de privacidad.

- **Sistemas autónomos.** Un tema de investigación actual en el campo de los DCSP es construir algoritmos que tengan cierto grado de autonomía. Aunque nosotros generalmente intentamos que los agentes actúen en nuestro favor, ellos deberían actuar sin intervención humana y tener

control sobre su estado y acciones. Esta propiedad mejoraría la flexibilidad, fiabilidad y robustez del sistema. Por ejemplo, cuando un agente que está ejecutando un algoritmo para DCSP cae, él debería recuperarse automáticamente sin la intervención humana. Además, los sistemas para DCSP deberían ser abiertos, permitiendo de este modo que el problema se fuera autodefiniendo dinámicamente. Los agentes podrían unirse o abandonar el contexto del problema y el espacio de búsqueda. Discusiones sobre la transparencia de los DCSP se presentan en [102], [25].

- **Heterogeneidad de los datos.** En DCSP la información es intrínsecamente distribuida y las fuentes de información a menudo son de diferentes organizaciones y tienen formatos, representaciones y semánticas de datos heterogéneos. Intercambiar información bajo estas condiciones resulta muy complicado. El reto es desarrollar esquemas de integración de datos que puedan sobrepasar estas dificultades usando métodos de traducción y estandarización.

A continuación vamos a presentar un sencillo ejemplo de CSP distribuido. Se trata del problema de generación de los horarios de enfermeras de un hospital; este problema se puede modelar como un CSP distribuido donde cada agente se encarga de la asignación de los horarios de las enfermeras de una planta, y se deben coordinar entre ellos para cumplir los requisitos de personal de la sala de urgencias.

1.3.1. Ejemplo de DCSP

El problema de generación de los horarios de enfermeras de un hospital puede ser formulado como un CSP distribuido. En este problema, hay un conjunto de enfermeras asignadas a cada una de las plantas del hospital, las cuales además deben cubrir sus turnos correspondientes en la sala de urgencias. El objetivo del problema es la generación de los horarios de las enfermeras de cada una de las plantas del hospital, los cuales son totalmente independientes

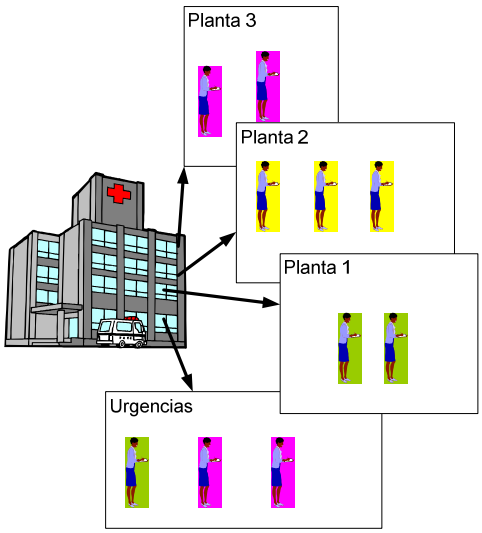


Figura 1.2: Problema de generación de los horarios de enfermeras.

a excepción de la sala de urgencias cuyos requerimientos de enfermeras deben estar cubiertos por las enfermeras de planta. De este modo cada agente se encargará de generar el horario de una de las plantas del hospital satisfaciendo todas las restricciones referentes a los requerimientos de personal del hospital y los requerimientos personales de cada enfermera. Además los agentes deben coordinarse entre ellos para satisfacer los requerimientos de personal de la sala de urgencias. En la Figura 1.2 se muestra una instancia concreta del problema de generación de horarios de enfermeras en un hospital. Tenemos un hospital con 3 plantas y una sala de urgencias. La planta 2 tiene 3 enfermeras y las plantas 1 y 3 tienen solo 2 enfermeras. De modo que tendremos un DCSP formado por 3 agentes, donde las restricciones de cada planta deben ser satisfechas por su correspondiente agente. Y además todos los agentes deben coordinarse para que en la sala de urgencias siempre haya 3 enfermeras.

1.4. Motivación y Contribución

Hoy en día, existen muchos problemas que pueden ser modelados como problemas de satisfacción de restricciones distribuidos donde el conocimiento del problema está dividido entre un conjunto de agentes; ejemplo de este tipo de problemas son: la coordinación de actividades en una red de sensores [21], la generación de horarios coordinados para los vehículos en un problema de logística del transporte [26], o la coordinación de reuniones entre un grupo de participantes a un evento [114]. Este tipo de problemas pueden ser resueltos mediante algoritmos explícitamente diseñados para realizar la búsqueda de una solución globalmente aceptada por todos los agentes, a la vez que se minimiza el proceso de comunicación entre ellos.

La mayoría del esfuerzo de investigación realizado en el campo de los DCSP se basa en la comunicación entre los agentes, y por lo general se asume que estos agentes trabajan con una sola variable. Sin embargo, en los problemas reales, es obvio que cada agente no maneja una única variable. Por lo tanto es importante tener en cuenta estas características y estudiar algoritmos para DCSP que sean capaces de manejar varias variables por agente.

Para suplir la limitación que implica el manejo de una única variable por agente, se han planteado dos métodos genéricos para la reformulación de los DCSP de manera que cada agente puede manejar múltiples variables [70]:

- **Método 1**: **Recompilación de *soluciones_parciales***. Cada agente encuentra todas las soluciones a su sub-problema. Una vez encontradas todas las soluciones, el problema puede ser reformalizado como un CSP distribuido en el que cada agente tiene una única variable cuyo dominio es el conjunto de soluciones de su sub-problema. Así los agentes pueden aplicar cualquier algoritmo como si manejasen una única variable. La desventaja de este método es que cuando el sub-problema es grande y complejo, encontrar todas las soluciones del sub-problema puede resultar una tarea muy compleja y en la práctica inviable.

- **Método 2**: **Descomposición de sub-problemas**. Cada agente crea

multiples agentes virtuales, cada uno de los cuales se corresponde con una de sus variables locales, y simula la actividad de los agentes virtuales. Por ejemplo, si el agente k tiene 2 variables locales X_i, X_j, se asume que existen dos agentes virtuales, cada uno asociado a una de las variables X_i o X_j. Entonces el agente k simula concurrentemente las actividades de los dos agentes virtuales. En este caso, cada agente no tiene que calcular todas las soluciones de su sub-problema. Sin embargo, como la comunicación entre los agentes suele ser más cara que los cálculos locales, es poco económico simular las actividades de los agentes virtuales sin distinguir entre las comunicaciones entre los agentes virtuales y las comunicaciones entre los agentes reales.

Aunque, aplicando alguno de estos métodos, la mayoría de los algoritmos que manejan una única variable por agente podrían aplicarse a situaciones en las que cada agente maneja multiples variables, los algoritmos que se obtienen no son ni eficientes ni tampoco escalables para grandes sub-problemas locales [70] debido a los requerimientos de tiempo y espacio para el cálculo de todas las soluciones en el primero de los métodos o debido a la sobrecarga de comunicación en el segundo método. Actualmente la mayoría de los algoritmos existentes para el manejo de multiples variables por agente, son adaptaciones de sus versiones clásicas en las que cada agente maneja una única variable.

Contribuciones

Esta tesis trata la necesidad de obtener modelos y técnicas que nos permitan manejar de manera eficiente problemas de satisfacción de restricciones distribuidos en los que cada agente debe manejar un conjunto amplio de variables. Para ello nos hemos centrado en dos características principales:

- El desarrollo de un nuevo algoritmo distribuido capaz de coordinar de manera asíncrona a agentes con múltiples variables.
- El desarrollo de métodos y técnicas heurísticas capaces de manejar con

mínimos requerimientos de espacio la información procedente de otros agentes. Esta información es utilizada para agilizar la búsqueda de soluciones_parciales de los sub-problemas de cada agente.

Además nuestro objetivo no se centra únicamente en el desarrollo de nuevos algoritmos distribuidos, sino que además proponemos la aplicación de estos nuevos métodos en entornos que por naturaleza no son distribuidos.

En un CSP distribuido, el conocimiento del problema (variables y restricciones) por lo general ya se encuentra distribuido entre un conjunto de agentes autónomos, y el objetivo de investigación es establecer un método para encontrar una solución al problema. Sin embargo, los problemas de satisfacción de restricciones que modelan problemas reales implican modelos con un gran número de variables y restricciones, generando densas redes de inter-relaciones. Esta clase de problemas puede ser manejada de manera global solo a costa de un exagerado coste computacional. Por esto, podría ser de gran ventaja particionar esta clase de problemas en un conjunto de sub-problemas más simples e interconectados entre sí, de tal modo que cada sub-problema pueda ser resuelto más fácilmente. El siguiente paso sería aplicar los algoritmos desarrollados para resolver los DCSPs para coordinar la resolución del conjunto de sub-problemas, obtenidos tras la partición, hasta alcanzar una solución al problema global. De este modo podemos resolver un problema centralizado mediante técnicas de resolución de problemas de satisfacción de restricciones distribuidos. Como ya hemos comentado anteriormente, por lo general, los algoritmos para tratar problemas de satisfacción de restricciones manejan una única variable por agente, por lo que, en este campo de investigación, no se plantea el tema de como distribuir el conjunto de variables del problema entre los agentes. Sin embargo, para problemas de gran tamaño, suponer que cada agente va a trabajar con una única variable no es factible, ya que se nos plantearía una red con miles de agentes. Esto nos motiva a la realización de esta tesis doctoral donde estudiaremos diversas técnicas para distribuir las variables entre los agentes.

Para poder aplicar los algoritmos distribuidos a problemas centralizados, en esta tesis planteamos diversos modos para distribuir el conjunto de variables entre los agentes:

1. Parece lógico que para reducir la comunicación entre los agentes una buena distribución sería por mínima conectividad. Para ello proponemos el uso de técnicas de particionamiento de grafos para dividir un CSP en varios sub-problemas menos complejos.

2. El siguiente método planteado se basa en la búsqueda de estructuras que debido a su topología requieren de un mínimo esfuerzo computacional para su resolución. De esta manera cada agente será capaz de resolver su sub-problema rápidamente, contribuyendo así a agilizar la búsqueda de la solución global. Para ello proponemos dividir el problema global en un conjunto de sub-problemas con estructura de árbol, cuyo coste de búsqueda es lineal.

3. Por último proponemos realizar el particionamiento de un problema centralizado mediante la identificación de entidades que son características de dicho problema, de manera que cada entidad represente un sub-problema.

1.4.1. Objetivos

El objetivo general de la tesis doctoral es extender la aplicabilidad de las técnicas de satisfacción de restricciones mediante el desarrollo de las técnicas correspondientes a problemas de satisfacción de restricciones distribuidos aplicables a complejos problemas locales, los cuales pueden resultar bien de una distribución inherente de un problema por naturaleza distribuido o bien de un particionamiento de un problema centralizado. Para alcanzar el objetivo general de la tesis doctoral desarrollaremos los siguientes objetivos específicos:

1. Identificación de restricciones distribuidas y jerarquías de variables que

determinen el modo más eficiente de distribución del problema. Más concretamente:

   a) Técnicas de particionamiento de grafos.

   b) Identificación de estructuras de árbol.

   c) Identificación de entidades dependientes del problema.

2. Cooperación y jerarquía en el proceso de resolución.

3. Diseño de algoritmos y heurísticas adecuadas para una resolución distribuida de las restricciones.

4. Aplicación y evaluación de las arquitecturas y de las técnicas desarrolladas en entornos reales: Resolución de problemas complejos en el contexto del trasporte ferroviario.

5. Aplicación y evaluación de las arquitecturas y de las técnicas desarrolladas en entornos ampliamente consensuados en la comunidad científica para la evaluación de CSP distribuidos.

1.5. Estructura del trabajo

El resto de la tesis doctoral está organizado como sigue:

- En el Capítulo 2 presentamos la modelización de un problema satisfacción de restricciones y las distintas técnicas existentes para la simplificación y resolución del problema de satisfacción de restricciones.

- En el Capítulo 3 presentamos las principales técnicas desarrolladas para la resolución de un problema de satisfacción de restricciones distribuido.

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

- En el capítulo 4 se presentan los 3 métodos propuestos para el particionamiento de un problema centralizado: particionamiento de grafos, identificación de árboles e identificación de entidades, indicando las características propias de cada método y justificando las ventajas de cada uno de ellos.

- En el Capítulo 5 presentamos un nuevo algoritmo para la resolución de problemas de satisfacción de restricciones distribuidos, el cual requiere una previa organización de los sub-problemas en una estructura de árbol. Además en este capítulo presentamos varios algoritmos para la resolución interna del sub-problema de cada agente.

- En el capítulo 6 presentamos varias heurísticas distribuidas que hacen uso de la información comunicada entre los agentes para podar el espacio de búsqueda de cada agente.

- En el capítulo 7 llevamos a cabo una evaluación empírica de los algoritmos y métodos de particionamiento propuestos en los capítulos anteriores. La evaluación se lleva a cabo tanto con problemas generados aleatoriamente, como con benchmarks de problemas de planificación ferroviaria.

- Finalmente en el capítulo 8 presentamos las conclusiones, posibles líneas de desarrollo que se generan a partir de este trabajo y las publicaciones relacionadas con la tesis.

Capítulo 2

Problemas de Satisfacción de Restricciones

En este Capítulo, introducimos el contenido necesario para poder comprender el resto del trabajo. Brevemente vamos a revisar varias técnicas de consistencia local, algoritmos de propagación de restricciones, algoritmos de búsqueda y heurísticas. Podemos encontrar un trabajo mucho más detallado en [111]. Además, podemos encontrar trabajos más breves de este área en [64], [17] y [76]. Este Capítulo está estructurado de a siguiente manera. En la sección 2.1 presentamos algunas definiciones, en la sección 2.2 propiedades de las técnicas de consistencia local que se han propuesto. En la sección 2.3 discutiremos los algoritmos de propagación de restricciones. En la sección 2.4 presentaremos los algoritmos más conocidos para el manejo de CSPs, que son los algoritmos de búsqueda sistemática. Finalmente en la sección 2.5 revisaremos algunas heurísticas tales como la heurística de ordenación de valores y la heurística de ordenación de variables.

2.1. Definiciones

A continuación vamos a definir algunos conceptos básicos para problemas de satisfacción de restricciones que son necesarios para la comprensión de esta tesis.

Definición 2.1.1. Un problema de satisfacción de restricciones (CSP) es una terna (X, D, C) donde:

- X es un conjunto de n variables $\{x_1, ..., x_n\}$.
- Para cada variable x_i $\in$X, D_i es el dominio de x_i. El dominio de una variable es el conjunto finito de todos los posibles valores que se le pueden asignar a esa variable.
- C es un conjunto finito de restricciones. Cada restricción k-aria (C_k) está definida sobre un conjunto de k variables $\{x_i, ..., x_j\}$ mediante el producto cartesiano $D_i \times ... \times D_j$, el cual representa combinaciones posibles de valores.

Definición 2.1.2. Una asignación de variables, también llamado instanciación, (x, a) es un par variable-valor que representa la asignación del valor a a la variable x. Una instanciación de un conjunto de variables es una tupla de pares ordenados, donde cada par ordenado (x, a) asigna el valor a a la variable x. Una tupla $((x_1, a_1), ..., (x_i, a_i))$ es *consistente* si satisface todas las restricciones formadas por variables de la tupla. Para simplificar la notación, sustituiremos la tupla $((x_1, a_1), ..., (x_i, a_i))$ por $(a_1, ..., a_i)$.

Definición 2.1.3. Una solución a un CSP es una asignación de valores a todas las variables de forma que se satisfagan todas las restricciones. Es decir, una solución es una tupla consistente que contiene todas las variables del problema. Una solución parcial es una tupla consistente que contiene algunas de las variables del problema. Básicamente los objetivos de un CSP son:

- Estudiar la consistencia del problema.
- Encontrar una solución.
- Enumerar varias o todas las soluciones.
- Encontrar los dominios mínimos de cada variable.
- Encontrar la mejor solución con respecto a algún criterio.

Si existe una solución al problema, entonces diremos que el problema es consistente. En caso contrario diremos que el problema es inconsistente.

2.1.1. Notación

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

General: El número de variables de un CSP lo denotaremos por n. La longitud del dominio finito de una variable x_i lo denotamos por d_i. El número de restricciones totales lo denotaremos por c. La *aridad* máxima de una restricción la denotaremos por k.

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

Dominios/Valores: : El dominio de una variable x_i lo denotamos por D_i. A los valores individuales de un dominio los representaremos mediante las primeras letras del alfabeto, por ejemplo, a, b, c, y al igual que en las variables también pueden ir seguidas de subíndices. La asignación de un valor a a una

variable x la denotaremos mediante el par (x, a). Como ya mencionamos en la definición (2.1.2) una tupla de asignación de variables $((x_1, a_1), ..., (x_i, a_i))$ la denotaremos por $(a_1, ..., a_i)$.

Restricciones: : Una restricción $k - aria$ entre las variables $\{x_1, ..., x_k\}$ la denotaremos por $C_{1..k}$. De esta manera, una restricción binaria entre las variables x_i y x_j la denotaremos por C_{ij}. Cuando los índices de las variables en una restricción no son relevantes, lo denotaremos simplemente por C. El conjunto de variables involucrados en una restricción $C_{i..k}$ lo representaremos por $X_{C_{i..k}}$. Las notaciones que surjan a lo largo del trabajo la definiremos cuando sean necesarias.

2.1.2. Restricciones

Vamos a dar algunas definiciones sobre restricciones y explicaremos algunas de las propiedades básicas. La aridad de una restricción es el número de variables que componen dicha restricción. Una restricción unaria es una restricción que consta de una sola variable. Una restricción binaria es una restricción que consta de dos variables. Una restricción no binaria (o $n - aria$) es una restricción que involucra a un número arbitrario de variables. Cuando nos referimos a un CSP binario queremos decir un CSP donde todas las restricciones son unarias o binarias. En cambio cuando nos referimos a un CSP no binario, nos referimos a un CSP donde alguna o todas las restricciones tienen una aridad mayor de dos.

Ejemplo. La restricción $x \leq 5$ es una restricción unaria sobre la variable x. La restricción $x_4 - x_3 \leq 3$ es una restricción binaria. La restricción $2x_1 - x_2 + 4x_3 \leq 4$ es una restricción no binaria.

Definición 2.1.4. Una tupla p de una restricción $C_{i..k}$ es un elemento del producto cartesiano $D_i \times .. \times D_k$. Una tupla p que satisface la restricción $C_{i..k}$

se le llama tupla permitida o válida. Una tupla p que no satisface la restricción $C_{i..k}$ se le llama tupla no permitida o no válida. De esta manera, verificando si una tupla dada es permitida o no por una restricción se llama chequeo de la consistencia o comprobación de la consistencia. Una restricción puede ser definida tanto extensionalmente por un conjunto de tuplas válidas o no válidas o intencionalmente como una función aritmética.

Ejemplo. Consideremos una restricción entre 4 variables x_1, x_2, x_3, x_4, con dominios $1, 2$, donde la suma entre las variables x_1 y x_2 es menor que la suma entre x_3 y x_4. Esta restricción puede ser representada *intencionalmente* mediante la expresión $x_1 + x_2 \leq x_3 + x_4$. Esta restricción también puede ser representada *extensionalmente* mediante el conjunto de tuplas permitidas $\{(1,1,1,1), (1,1,1,2), (1,1,2,1), (1,1,2,2), (2,1,2,2), (1,2,2,2), (2,2,2,2)\}$, o mediante el conjunto de tuplas no permitidas $\{(1,2,1,1), (2,1,1,1), (2,2,1,1), (2,2,1,2), (2,2,2,1)\}$.

Grafo de Restricciones

Un CSP binario puede ser representado 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. Para cada arista que une el nodo i con el nodo j correspondiente a la restricción C_{ij} entre las variables x_i y x_j, hay una arista que une el nodo j con el nodo i correspondiente a la restricción C_{ij} tal que ambas restricciones tienen el mismo conjunto de tuplas permitidas. Es decir, las dos restricciones son simétricas. En la Figura 2.1 presentamos un ejemplo de grafo de restricciones binarias que representa a un CSP con 4 variables x_1, x_2, x_3, x_4 y 4 restricciones C_{12}, C_{13}, C_{14} y C_{34}.

Por simplicidad, si hay una arista que une las variables x_i y x_j, entonces no mostramos su arista simétrica que une las variables x_j y x_i. De esta manera, una arista conectando dos nodos x_i y x_j se representa como una arista

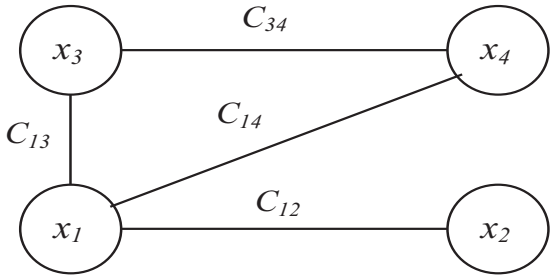


Figura 2.1: Grafo de restricciones binarias

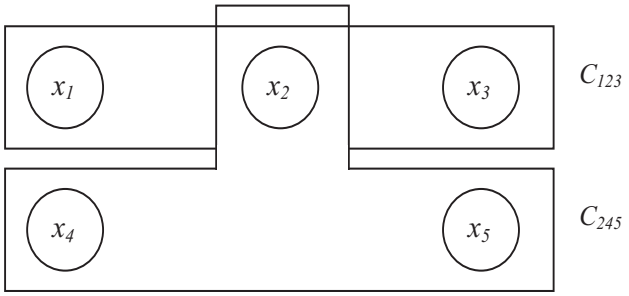


Figura 2.2: Ejemplo de restricciones no binarias

no dirigida. Un CSP no binario se puede representar mediante un hipergrafo donde los nodos representan las variables, y las hiperaristas representan las restricciones no binarias. Una hiperarista conecta los nodos que componen la correspondiente restricción no binaria. Por ejemplo, en el grafo de restricciones de la Figura 2.2 hay dos restricciones no binarias, una compuesta por las variables $\{x_1, x_2, x_3\}$, y la otra por las variables $\{x_2, x_4, x_5\}$.

La vecindad de una variable x en un CSP binario es el conjunto de variables en el grafo restringido que son adyacentes a x. En un CSP no binario, la vecindad de una variable x está formada por el conjunto de variables que están conectadas por un hiperarco con x. El grado de una variable x en un CSP es el número de restricciones en las que x participa. Para un CSP binario, esto es lo mismo que el número de variables que son adyacentes a x en el grafo.

Árbol de Búsqueda

Las posibles combinaciones de la asignación de variables en un CSP genera un espacio de búsqueda que puede ser visto como un árbol, llamado árbol de búsqueda. La búsqueda mediante backtracking en un CSP corresponde a la tradicional 'exploración primero en profundidad' en el árbol de búsqueda. Si asumimos que el orden de las variables es estático y no cambia durante la búsqueda entonces un nodo en el nivel k del árbol de búsqueda representa un estado donde las variables $x_1, ..., x_k$ están asignadas y el resto $x_{k+1}, ..., x_n$ no lo están. Nosotros podemos asignar cada nodo en el árbol de búsqueda con la tupla consistente de todas las asignaciones llevadas a cabo. La raíz del árbol de búsqueda representa la tupla vacía, donde ninguna variable tiene asignado valor alguno. 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 es solución del problema. Un nodo del árbol de búsqueda es consistente si la asignación parcial actual es consistente, o en otras palabras, si la tupla correspondiente a ese nodo es consistente. 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 llama 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 les llama nodos superficiales. Los nodos próximos a las hojas del árbol de búsqueda se le llaman nodos profundos. En la Figura 2.3 presentamos un

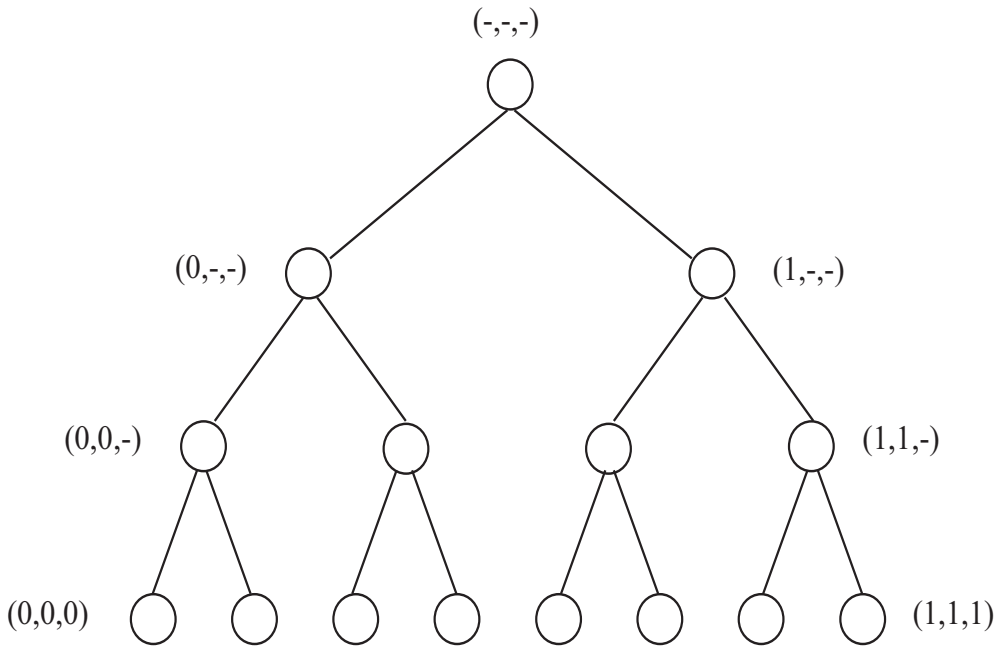


Figura 2.3: Árbol de búsqueda

ejemplo de árbol de búsqueda.

La figura muestra un ejemplo con tres variables y cuyos dominios son 0 y 1. Cada variable corresponde a un nivel en el árbol. En el nivel 0 no se ha hecho ninguna asignación, por lo que se representa mediante la tupla $(-, -, -)$. Los demás nodos corresponden a tuplas donde al menos se ha asignado una variable. Por ejemplo el nodo hoja que se encuentra a la derecha en el árbol de búsqueda $(1, 1, 1)$, 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 llama variable actual. Las variables correspondientes a niveles menos profundos se llaman variables pasadas. Las variables restantes que se instanciarán más tarde se llaman variables futuras.

2.2. Técnicas de Consistencia Local

Como veremos en la sección 2.4, los algoritmos de búsqueda sistemática para la resolución de CSPs tienen como base la búsqueda basada en backtracking. Sin embargo, esta búsqueda sufre con frecuencia una explosión combinatoria en el espacio de búsqueda, y por lo tanto no es por sí solo un método suficientemente eficiente para resolver CSPs. Una de las principales dificultades con las que nos encontramos en los algoritmos de búsqueda es la aparición de inconsistencias locales que van apareciendo continuamente [71]. Las inconsistencias locales son valores individuales o combinación de valores de las variables que no pueden participar en la solución porque no satisfacen alguna propiedad de consistencia. Por ejemplo, si el valor a de la variable x es incompatible con todos los valores de una variable y que está ligada a x mediante una restricción, entonces a es inconsistente con respecto a la propiedad local de consistencia de arco o arco-consistencia[1]. Por lo tanto si forzamos alguna propiedad de consistencia A podemos borrar todos los valores que son inconsistentes con respecto a la propiedad A. Esto no significa que todos los valores que no pueden participar en una solución sean borrados. Puede haber valores que son consistentes con respecto a A pero son inconsistentes con respecto a cualquier otra consistencia local B. Sin embargo, consistencia *Global* significa que todos los valores que no pueden participar en una solución son eliminados.

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

Ejemplo. 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:

[1] Arco-Consistencia lo definiremos más adelante

$x \neq y$ y $y < z$. Si asumimos que la búsqueda de backtracking trata de instanciar las variables en el order x, y, z entonces probará todas las posibles combinaciones de valores para las variables antes de descubrir que no existe solución alguna. Si miramos la restricción entre y y z podremos ver que no hay ninguna combinación de valores para las dos variables que satisfagan la restricción. Si el algoritmo pudiera identificar esta inconsistencia local antes, se evitaría un gran esfuerzo de búsqueda.

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

A continuación vamos a definir varias técnicas de consistencia local que se han propuesto en la literatura. Tales definiciones están referidas a CSPs binarios debido a que las técnicas de consistencia para CSPs no binarios no se han estudiado con profundidad y además no se han definido formalmente.

2.2.1. Consistencia de Nodo

La consistencia local más simple de todas es la *consistencia de nodo*. Forzar esta consistencia nos asegura que todos los valores de una variable satisfacen todas las restricciones unarias sobre esa variable.

Ejemplo. Consideremos una variable x en un problema con dominio continuo $[2, 15]$ y la restricción unaria $x \leq 7$. La consistencia de nodo eliminará el intervalo $]7, 15]$ del dominio de x. En la Figura 2.4 mostramos el resultado de aplicar nodo-consistencia a la variable x.

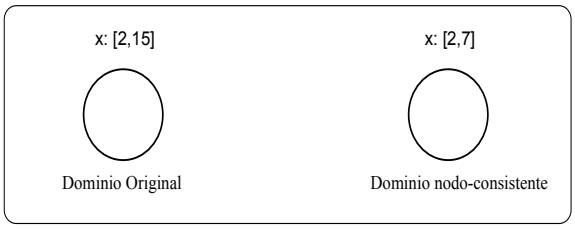


Figura 2.4: Consistencia de nodo, nodo-consistencia

2.2.2. Consistencia de Arco

La consistencia local más utilizada es la *consistencia de arco o arco-consistencia* [71]. Un problema binario es arco-consistente si para cualquier par de variables restringidas x_i y x_j, para cada valor $a \in D_i$ hay al menos un valor $b \in D_j$ tal que las asignaciones (x_i, a) y (x_j, b) satisfacen la restricción entre x_i y x_j. Cualquier valor en el dominio D_i de la variable x_i que no es arco-consistente puede ser eliminado de D_i ya que no puede formar parte de ninguna solución. El dominio de una variable es arco-consistente si todos sus valores son arco-consistentes.

Según la definición dada por Debruyne y Bessiére [29], un CSP binario es *arco-consistente* si y solamente si $\forall a \in D_i, \forall x_j \in X$, con $C_{ij} \in C, \exists b \in D_j$ tal que b es un soporte para a en C_{ij}.

2.2.3. Consistencia de Senda

La consistencia de senda [80] (Path-consistente) es un nivel más alto de consistencia local que arco-consistencia. La consistencia de senda requiere para cada par de valores a y b de dos variables x_i y x_j, que la asignación de a a x_i y de b a x_j satisface 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. Montanari demostró que un CSP satisface la consistencia de senda si y sólo si todos los caminos de longitud dos cumplen la consistencia de senda [80]. Cuando un problema satisface la consistencia de senda y además es nodo-consistente y arco-consistente se dice que satisface

fuertemente la consistencia de senda (strongly path-consistent).

Según la definición dada por Debruyne y Bessiére [29], un CSP binario es *path-consistente* si y solamente si $\forall x_i, x_j \in X, (x_i, x_j)$ es path-consistente. Un par de variables (x_i, x_j) es *path-consistente* si y solo si $\forall (a, b) \in C_{ij}, \forall x_k \in X, \exists c \in D_k$ por lo que c está soportado por a en c_{ik} y c está soportado por b en c_{jk}.

El estudio de la consistencia de senda no se usa ampliamente debido a su alto coste temporal y espacial, y también debido al hecho de que puede modificar el grafo de restricciones añadiendo nuevas restricciones binarias. Sin embargo es una importante consistencia en CSPs con ciertas propiedades como en CSPs temporales [33].

2.2.4. 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. Además la consistencia de arco no puede definirse 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. La consistencia de borde es una noción genérica que puede ser aplicado a CSPs de cualquier aridad. Sin embargo la consistencia de borde es más débil que la consistencia de arco. veamoslo en el siguiente ejemplo.

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 arco-consistentes. Sin embargo, el problema no es arco-consistente porque el valor 0 para y no es arco-consistente. No hay valores en el dominio de x que satisfagan la restricción cuando y es 0.

2.3. Algoritmos de Propagación de Restricciones

Como hemos comentado en la sección anterior, se han propuesto en la literatura muchos niveles de consistencia local como consistencia de arco o consistencia de senda. Los algoritmos que alcanzan tales niveles de consistencia eliminan pares variable-valor inconsistentes, es decir nodos del árbol de búsqueda, que no pueden participar en ninguna solución. A tales algoritmos se les llama *algoritmos de propagación de restricciones* o *algoritmos de filtrado*. Aunque aplicando los algoritmos de propagación de restricciones 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 preproceso para reducir la talla de los CSPs y además durante la búsqueda para podar el espacio de búsqueda. Los algoritmos de propagación de restricciones se han estudiado ampliamente como formas de manejar CSPs. En esta sección revisamos algunos trabajos previos sobre algoritmos de propagación de restricciones.

La mayoría del trabajo realizado sobre problemas de propagación de restricciones se han centrado en la arco-consistencia, porque alcanzar la consistencia de arco es una manera 'barata' de eliminar algunos valores que no pueden pertenecer a ninguna solución. Se ha demostrado que los algoritmos de arco-consistencia son muy efectivos cuando se aplican como técnica de preproceso y también durante la búsqueda. Hay un gran número de algoritmos propuestos en la literatura que pueden alcanzar la consistencia de arco sobre un grafo de restricciones binarias.

Mackworth describe los algoritmos básicos AC-1, AC-2 y AC-3 [71]. El coste temporal en el peor caso en el algoritmo AC-3, el cual es el mejor entre los tres, es de $O(cd^3)$ donde d es la talla del dominio más grande y c es el número de restricciones del grafo. La complejidad espacial de AC-3 es $O(c + nd)$ donde n es el número de variables. Este algoritmo ha sido mejorado por

muchos autores, dando lugar a nuevas versiones de AC-3. Bessiere y Regín [23] mejoran el comportamiento de AC-3 mediante dos algoritmos AC2000 y AC2001 que presentan una notable mejoría en cuanto a eficiencia. Mohr y Henderson introdujeron el algoritmo AC-4 [79] el cual tenía un óptimo peor caso de $O(cd^2)$, aunque con frecuencia se alcanzaba un comportamiento en el peor caso igual al de AC-3. Además la complejidad espacial es $O(cd^2)$. Dos algoritmos AC-5 se propusieron para reducir la complejidad temporal de AC-4 a $O(kd)$ en tipos específicos de restricciones. Así sucesivamente se fueron generando versiones de los algoritmos de arco-consistencia que mejoraban las versiones anteriores.

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 c_{ij} , exista un valor b en el dominio D_j que satisfaga la restricción $(x_i c_{ij} x_j)$. De este modo, la arco-consistencia direccional es sólo útil para garantizar que los valores de las variables de menor orden son consistentes con los valores de las variables de mayor orden.

2.4. Algoritmos de Búsqueda Sistemática

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

En la literatura se han desarrollado muchos algoritmos de búsqueda completa para CSPs. Algunos ejemplos son: 'backtracking cronológico', 'backjumping' [42], 'conflict-directed backtracking' [86], 'backtracking dinámico' [44], 'forward-checking' [48], 'minimal forward checking' [35], algoritmos híbridos como forward checking con conflict-directed backtracking [86] y 'manteniendo arco-consistencia' (MAC) [42] [91].

2.4.1. Algoritmo de Backtracking Cronológico

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

El backtracking cronológico es un algoritmo muy simple pero es muy ineficiente. El problema es que tiene una visión local del problema. Sólo chequea restricciones que están formadas por la variable actual y las pasadas, e ignora

la relación entre la variable actual y las futuras. Además, este algoritmo es ingenuo en el sentido de que no 'recuerda' las acciones previas, y como resultado, puede repetir la misma acción varias veces innecesariamente. Para ayudar a combatir este problema, se han desarrollado algunos algoritmos de búsqueda mas robustos. Estos algoritmos se pueden dividir en algoritmos *look-back* y *look-ahead*.

2.4.2. Algoritmos Look-Back

Los algoritmos look-back tratan de explotar la información del problema para comportarse más eficientemente en las situaciones sin salida. Al igual que el backtracking cronológico, los algoritmos look-back llevan a cabo el chequeo de la consistencia *hacia atrás*, es decir, entre la variable actual y las pasadas. Veamos algunos algoritmos look-back.

- *Backjumping*(BJ) [42] es un algoritmo para CSPs parecido al backtracking cronológico excepto que se comporta de una manera más inteligente cuando encuentra situaciones sin salida. En vez de retroceder a la variable anteriormente instanciada, BJ salta a la variable más profunda (más cerca de la raiz en el árbol de búsqueda) x_j que esta en conflicto con la variable actual x_i donde $j < i$. Decimos que una variable instanciada x_j está en conflicto con una variable x_i si la instanciación de x_j evita uno de los valores en x_i (debido a la restricción entre x_j y x_i). Cambiar la instanciación de x_j puede hacer posible encontrar una instanciación consistente de la variable actual.

- *Conflict-directed Backjumping*(CBJ) [86] tiene un comportamiento de salto hacia atrás más sofisticado que BJ. Cada variable x_i tiene un *conjunto conflicto* formado por las variables pasadas que están en conflicto con x_i. En el momento en el que 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 complicadas para almecenar los conjuntos conflictos.

- *Learning* [40] es un método que almacena las restricciones implícitas que son derivadas durante la búsqueda y las usa para podar el espacio de búsqueda. Por ejemplo cuando se alcanza una situación sin salida en la variable x_i entonces sabemos que la tupla de asignaciones $(x_1, a_1), ..., (x_{i-1}, a_{i-1}))$ nos lleva a una situación sin salida. Así, nosotros podemos *aprender* que una combinación de asignaciones para las variables $x_1, ..., x_{i-1}$ no está permitida.

2.4.3. Algoritmos Forward Checking

Como ya indicamos anteriormente, los algoritmos look-back tratan de reforzar el comportamiento de BT mediante un comportamiento más inteligente cuando se encuentran en situaciones sin salida. Sin embargo, todos ellos todavía llevan a cabo el chequeo de la consistencia solamente hacia atrás, ignorando las futuras variables. Los algoritmos *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 esa manera, las situaciones sin salida se pueden identificar antes y además los valores inconsistentes se pueden descubrir y podar para las variables futuras.

Forward Checking para restricciones binarias

- *Forward checking* (FC) [48] es uno de los algoritmos look-ahead más comunes. En cada etapa de la búsqueda, FC chequea hacia adelante la asignación actual contra todos los valores de las futuras variables que

están restringidas con la variable actual. Los valores de las futuras variables que son inconsistentes con la asignación actual son temporalmente eliminados de sus dominios. Si el dominio de una variable futura se queda vacío, la instanciación de la variable actual se deshace y se prueba con un nuevo valor. Si ningún valor es consistente, entonces se lleva a cabo el backtracking cronológico. FC garantiza que en cada etapa la solución parcial actual es consistente con cada valor de cada variable futura. Además cuando se asigna un valor a una variable, solamente se chequea hacia adelante con las futuras variables 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 forward checking se puede ver como aplicar un simple paso de arco-consistencia sobre cada restricción que involucra la variable actual con una variable futura después de cada asignación de variable.

Forward checking se ha combinado con algoritmos look-back para generar algoritmos *híbridos* [86]. Por ejemplo, *forward checking* con *conflict-directed backjumping* (FC-CBJ) [86] es un algoritmo híbrido que combina el movimiento hacia adelante de FC con el movimiento hacia atrás de CBJ, y de esa manera contiene las ventajas de ambos algoritmos.

- *Minimal forward checking* (MFC) [35] es una version de FC que retrasa llevar a cabo todo el chequeo de la consistencia de FC hasta que es 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 limpieza de dominios. Para hacer esto es suficiente chequear la asignación actual contra los valores de cada variable futura hasta que se encuentra una que es consistente. Después, si el algoritmo ha retrocedido, vuelve atrás y lleva a cabo 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 % [35].

- Bacchus propuso nuevas versiones de forward checking [15]. Estas versiones están basadas en la idea de desarrollar un mecanismo de poda de dominios que elimina valores no sólo en el nivel actual de búsqueda, sino también en cualquier otro nivel. Bacchus primero describe una plantilla genérica para implementar varias versiones de forward checking y después describe cuatro instanciaciones de esa plantilla devolviendo nuevos algoritmos de forward checking. Los primeros dos algoritmos, *extended forward checking* (EFC) y EFC-, tienen la habilidad de podar futuros valores que son inconsistentes con asignaciones hechas antes de la asignación actual pero que no habían sido descubiertas. Los otros dos algoritmos, *conflict based forward checking* (CFFC) y CFFC-, están basados en la idea de los conflictos, al igual que CBJ y learning para reforzar a FC. CFFC y CFFC- almacenan los conjuntos conflicto y los usan para podar lo valores 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 conflicto. Esto permite saltar más lejos que CBJ.

2.4.4. Mantenimiento de la Arco-consistencia

El algoritmo de mantenimiento de la arco-consistencia (MAC) [42] [91] es un algoritmo conocido que realiza más trabajo cuando mira hacia adelante después de una instanciación o eliminación de valores. Cuando se intenta la asignación de una variable MAC aplica arco-consistencia al subproblema formado por todas las futuras variables. Esto significa que además del trabajo que FC realiza, MAC chequea también las variables futuras. 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. Entonces se prueba con un nuevo valor de 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 cronológico.

El trabajo extra que MAC realiza al aplicar arco-consistencia puede eliminar más valores de las futuras variables y como consecuencia podar más el árbol de búsqueda que FC. Hay varias versiones de MAC dependiendo del algoritmo de arco-consistencia empleado. Por ejemplo, MAC-3 mantiene arco-consistencia utilizando un algoritmo AC-3, mientras que MAC-7 usa un algoritmo AC-7.

MAC, al igual que FC, se ha combinado con algoritmos look-back para obtener algoritmo híbridos. Por ejemplo MAC-CBJ [87] 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 [15], 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.

MAC se ha generalizado a un algoritmo que mantiene arco-consistencia generalizada (MGAC) o también llamado Real-Full-Look-Ahead (RFLA). 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 [89].

Ha habido algunos estudios donde MGAC, o versiones de el, se utilizaron para restricciones no binarias arbitrarias. En [112] se presentan experimentos sobre problemas de planificación utilizando un algoritmo MGAC complementado con *conflict-direct backjumping* (MGAC-CBJ). En [15] se presentan experimentos realizados sobre problemas de planificación del mundo de bloques utilizando una versión no binaria del algoritmo CFMAC.

2.5. Heurísticas

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

2.5.1. Ordenación de Variables

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

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. La intuición es tratar de asignar lo antes posible las variables más restringidas y de esa manera identificar las situaciones sin salida lo antes posible y así reducir el número de vueltas atrás.

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 with* (MW) [39] impone en primer lugar un orden total sobre las variables, de forma que el orden tiene la mínima anchura, y entonces selecciona las variables en base a ese orden. La *anchura* de la variable x es el número de variables que están antes de x, de acuerdo a un orden dado, y que son adyacentes a x. La anchura de un orden es la máxima anchura de todas las variables bajo ese orden. La anchura de un grafo de restricciones es la anchura mínima de todos los posibles ordenes. Después de calcular la anchura de un grafo de restricciones, las variables se ordenan desde la última hasta la primera en anchura decreciente. Esto significa que las variables que están al principio de la ordenación son las más restringidas y las variables que están al final de la ordenación son las menos restringidas. Asignando las variables más restringidas al principio, las situaciones sin salida se pueden identificar antes y además se reduce el número de backtracking.

La heurística *maximun degree* (MD) [32] ordena las variables en un orden decreciente de su grado en el grafo de restricciones. El *grado* de un nodo se define como el número de nodos que son adyacentes a él. Esta heurística también tiene como objetivo encontrar un orden de anchura mínima, aunque no lo garantiza.

La heurística *maximun cardinality* (MC) [88] 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 [32] 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. Esto es porque lo que 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 se basa en el principio de *primer fallo* (FF) [48] que sugiere que *para tener éxito deberíamos intentar primero donde sea más probable que falle*. De esta manera las situaciones sin salida pueden identificarse antes y además se ahorra espacio de búsqueda. De acuerdo con el principio de FF, en cada paso, seleccionaríamos la variable más restringida. La heurística FF también conocida como heurística *minimun remaining values* (MRV), trata de hacer lo mismo seleccionando la variable con el dominio más pequeño. Esto se basa en la intuición de que si una variable tiene pocos valores, entonces es más difícil encontrar un valor consistente. Cuando se utiliza MRW 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 MRW 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 a asignarse 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 [63] se introduce una heurística que trata de seleccionar la variable más restringida evaluando lo restringidos que están los valores 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). Entonces estas evaluaciones se combinan para producir una estimación de cuan restringida esta x. La variable más restringida se selecciona.

Geelen propuso una heurística similar basada en el conteo del número de valores soporte para cada valor de una variable no instanciada [43]. 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 por el contrario es una heurística cara de calcular. Su complejidad temporal para una asignación de 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. 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.

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 [63] Keng y Yun proponen una variación de la idea anterior. De acuerdo a su heurística, cuando se cuenta el número de valores incompatibles para una futura variable x_k, éste se divide por la talla del dominio de x_k. Esto da el porcentaje de los valores útiles que pierde x_k debido al valor a que actualmente estamos examinando. De nuevo los porcentajes se añaden para todas las variables futuras y se selecciona el valor más bajo que se obtiene en todas las sumas.

Geelen [43] propuso una heurística de ordenación de valores a la cual llamó *promise*. Para cada valor a de la variable x 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 de que este valor se haya asignado a la variable actual. La promesa de cada valor representa una cota superior del número de soluciones diferentes que pueden existir después de que el valor se asigne a la variable.

En [41] se describen tres heurísticas de ordenación de valores dinámicos inspirados por la intuición de que un subproblema es más probable que tenga solución si no tiene variables que sólo tengan un valor en su dominio. La primera heurística, llamada heurística *max-domain-size* selecciona el valor de la variable actual que crea el máximo dominio mínimo en las variables futuras. La segunda heurística, llamada *weighted-max-domain-size* es una mejora de la primera. Esta heurística especifica una manera de romper empates basada en el número de futuras variables que tiene una talla de dominio dado. Por ejemplo, si un valor a_i deja cinco variables futuras con dominios de dos elementos, y otro valor a_j deja siete variables futuras también con dominios de dos elementos, en este caso se selecciona el valor a_i. La tercera heurística, llamada *point-domain-size*, asigna un peso (unidades) a cada valor de la variable actual dependiendo del número de variables futuras que se quedan con ciertas tallas de dominios.

Por ejemplo, para cada variable futura que se queda con un dominio de talla uno debido a la variable a_i, se añaden 8 unidades al peso de a_i. De esta manera se selecciona el valor con el menor peso. Estas heurísticas parecen ser muy a medida y los resultados experimentales en [41] muestran que la heurística *min-conflict* supera a estas tres heurísticas.

2.6. Conclusiones

En este capítulo se ha efectuado una revisión de los principales conceptos y técnicas relacionadas con la metodología CSP. Se destaca la gran actividad en un área de elevado interés tanto científico como aplicado.

Capítulo 3

Algoritmos para CSP Distribuidos

En este capítulo presentamos los problemas de satisfacción de restricciones distribuidos y el estado del arte referente a los principales algoritmos para la resolución de DCSPs. Primero se presentan las versiones clásicas de los principales algoritmos donde se asume que cada agente maneja únicamente una variable. A continuación se presentan los algoritmos para la resolución de DCSP donde se asume que cada agente puede manejar varias variables. Finalmente se presentan los principales algoritmos para la resolución de problemas de optimización de restricciones distribuidos.

3.1. Definición de DCSP

A continuación introducimos varias definiciones que nos permiten hacer una correcta descripción de un problema de satisfacción de restricciones distribuido.

Definición 3.1.1. Un ***CSP distribuido*** (**DCSP**) es un CSP con sus variables y restricciones distribuidas entre "*agentes*" autónomos [119]. Cada "*agente*" conoce los dominios de sus variables y un conjunto de restricciones, e intenta determinar los valores de sus variables. Además, existen un conjunto

de restricciones entre los agentes que también deben ser satisfechas por los valores asignados a las variables. En la Figura 3.1 podemos ver un ejemplo de un *CSP distribuido* compuesto por dos "*agentes*" (A_1 y A_2).

Definición 3.1.2. Se define como ***inter-restricción*** a cada restricción que relaciona dos variables de diferentes "*agentes*". A cada restricción que relaciona dos variables que pertenecen al mismo "*agente*" la llamaremos ***intra-restricción***. En la Figura 3.1, la arista 1-3 es una *inter-restricción* y la arista 1-2 es una *intra-restricción*.

Definición 3.1.3. Denotamos por:

- V_{agente_id} al conjunto de variables que pertenecen al *agente* con identificador *id*. En la Figura 3.1, $V_2 = \{3, 5, 7\}$;

- V_{nogood} al conjunto de variables incluidas en un mensaje *Nogood*.

De manera análoga a los CSPs, los DCSP también pueden clasificarse en dos categorías: los algoritmos de búsqueda sistemática distribuida y los algoritmos de búsqueda local distribuida. En las siguientes secciones presentamos los principales algoritmos de ambas categorías, así como las revisiones que se han hecho de estos algoritmos para adaptarlos al manejo de varias variables por agente.

Igualmente, cuando en un DCSP, además de buscar una solución, lo que se pretende es que la solución sea óptima (de acuerdo al criterio de cierta función objetivo), entonces, se deben usar algoritmos específicos para problemas de optimización de restricciones distribuidos (DCOP - Distributed Constraint Optimization Problem). Este tipo de problemas no entran dentro del ámbito de estudio de esta tesis, pero al final del capítulo haremos una breve introducción de los principales algoritmos de resolución de DCOP.

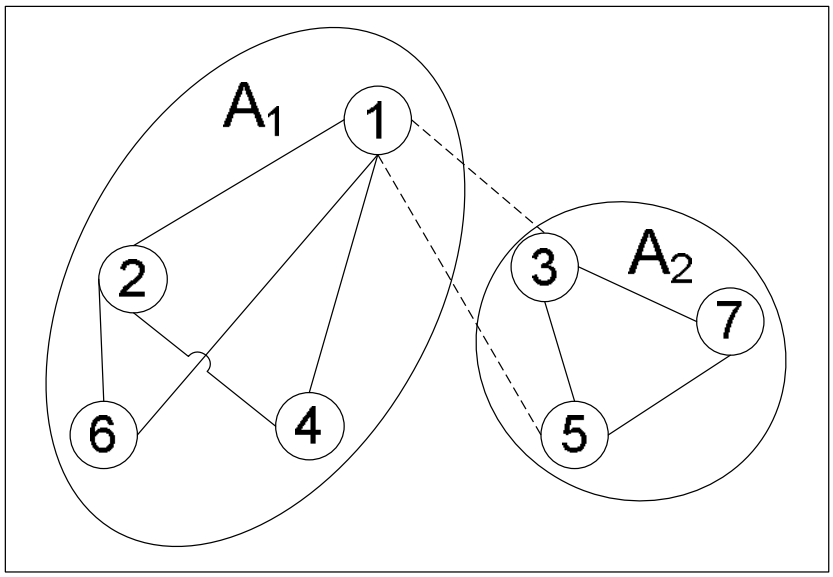


Figura 3.1: Ejemplo CSP Distribuido.

3.2. Algoritmos Distribuidos para problemas de satisfacción de restricciones

En este apartado presentamos los principales algoritmos desarrollados para la resolución de problemas de satisfacción de restricciones distribuidos, tanto de búsqueda sistemática como de búsqueda local. Todos ellos asumen por simplificación que cada agente posee únicamente una variable.

3.2.1. Algoritmos distribuidos de búsqueda sistemática

Synchronous Backtracking

El *Algoritmo de Backtracking Cronológico* (BT) presentado en el apartado 2.4.1 se puede convertir en un algoritmo síncrono distribuido para CSPs. Para ello los agentes se van comunicando secuencialmente las asignaciones parciales de sus variables de acuerdo a un orden previamente establecido. De esta manera cada agente recibe una *solución_parcial* con la asignación de las variables de

los agentes previos; entonces el agente intenta instanciar su variable de acuerdo a la *solución_parcial* recibida, si lo consigue envía la nueva *solución_parcial* al siguiente agente; si no encuentra una asignación válida envía un mensaje de *backtracking* al agente previo. Una descripción de este algoritmo se puede ver en [117].

Este algoritmo no explota las ventajas de la ejecución en paralelo ya que en cada instante de tiempo solo hay un agente en acción. Básicamente se trata de un algoritmo de backtracking centralizado, pero en el que el control de la asignación va pasando por diferentes agentes a lo largo de la ejecución. Simplemente adquiere cierta ventaja en cuanto a la privacidad ya que todos los agentes no necesitan conocer todas las restricciones.

Asynchronous Backtracking: ABT

Asynchronous backtracking (ABT) [116] es uno de los algoritmos pioneros para la resolución de CSPs distribuidos. En este algoritmo el orden de los agentes, o prioridad, está determinado.

Durante el proceso de ejecución del algoritmo, cada agente asigna un valor a su variable y envía un mensaje *ok?* a sus vecinos con menor prioridad. Cada agente mantiene una *lista* con las asignaciones del resto de sus vecinos de mayor prioridad. Cuando un agente recibe un mensaje *ok?* actualiza su *lista* de asignaciones y comprueba si el valor de su variable sigue siendo consistente. Un agente cambia el valor de su variable cuando su valor es inconsistentes con los valores de la *lista* de asignaciones. Cuando encuentra una asignación válida vuelve a enviar un mensaje *ok?* a sus vecinos menos prioritarios. Si un agente no puede encontrar ninguna asignación válida consistente con la *lista* de asignaciones, le envía un mensaje *nogood* al agente de menor prioridad con alguna variable incluida en el *nogood* para que cambie el valor de su variable, elimina de su *lista* la asignación de la variable del agente al que envía el *nogood* y vuelve a comprobar la consistencia de su asignación. El mensaje *nogood* debe contener la lista de asignaciones del agente que la envía, para que el agente

que lo recibe compruebe si coincide con la suya, ya que debido a que todos los agentes actúan asíncrona y concurrentemente, la lista de asignaciones puede estar obsoleta. Cuando un agente recibe un mensaje *nogood* lo almacena como una nueva restricción. Si el mensaje contiene el valor de una variable que no pertenece a ninguno de sus agentes vecinos, le pide a este agente que se convierta en su vecino y almacena en su *lista* el valor de la variable del nuevo vecino, luego comprueba la consistencia de su estado; si es consistente le manda un mensaje *ok?* al agente que le envió el *nogood* para que actualice su *lista* de asignación; si no es consistente busca una nueva solución y envía un mensaje *ok?* a todos sus vecinos menos prioritarios, si no encuentra solución envía un *nogood* a su vecino con menor prioridad con alguna variable en el *nogood*.

Este algoritmo puede ser mejorado de diversas formas:

- Los agentes deben leer todos los mensajes antes de realizar algún cálculo. La idea es que un agente no realice ningún chequeo de consistencia hasta que todos los mensajes que ha recibido son leídos y su *lista* de asignaciones es actualizada. Esto produce que el algoritmo sea mucho más eficiente en el caso de que no se produzcan retrasos en los mensaje o los retrasos sean de un tiempo fijo, ya que el agente tiende a recibir todos los mensajes de sus vecinos a la vez. Sin embargo, cuando los retrasos de los mensajes son aleatorios no se alcanza ninguna mejora ya que los agentes tienden a responder mensaje a mensaje.

- Crear un *nogood* de tamaño mínimo que incluya solo las variables de los agentes que crean inconsistencia. Esto mejora el rendimiento del algoritmo. El proceso se corresponde con el algoritmo *Distributed Dynamic Backtracking* [22].

El algoritmo presenta varios inconvenientes que dificultan su uso en problemas reales. En el peor caso, los agentes almacenan (por duplicado) el espacio de búsqueda completo. Además, toda esta información almacenada debe ser comprobada cada vez que se instancia una variable, por lo que el coste de las asignaciones va creciendo a medida que vamos almacenando *nogoods*. Aunque

el problema original esté formado sólo por restricciones binarias, cuando almacenamos *nogoods* puede ser que se creen restricciones de mayor aridad, lo cual provoca un mayor coste en la comunicación, el chequeo y la memoria.

Distributed Backtracking: DIBT

A continuación presentamos el algoritmo Distributed Backtracking (DIBT), desarrollado por Youssef Hamadi, Christian Bessière y Joël Quinqueton [46].

El primer paso para ejecutar el algoritmo es determinar el orden de los agentes, para ello cada agente calcula su posición y el conjunto de agentes que son sus padres o hijos en función de la heurística elegida.

A continuación cada agente instancia su variable con respecto a las restricciones de sus padres (inicialmente no se tienen en cuenta porque el agente desconoce el valor que su padre está asignando), después envía un mensaje *info* a sus hijos con el valor asignado. Si no encuentra ningún valor compatible entonces envía un mensaje *back* a su padre más cercano, este mensaje contiene el conjunto de valores que están en la actual vista del agente para que el padre compruebe si la información es obsoleta.

Cuando un agente recibe un mensaje *info*, comprueba si su asignación sigue siendo compatible; si no lo es, la cambia e informa a sus hijos, o envía un mensaje *back* si no encuentra un valor compatible.

Cuando un agente recibe un mensaje *back*, comprueba si la información es obsoleta; si no lo es intenta hacer una nueva asignación, si lo consigue envía un mensaje *info* a sus hijos, y si no un mensaje *back* a su padre más cercano.

Este algoritmo permite un alto nivel de asincronismo ya que permite búsquedas simultáneas en distintos puntos de la red.

Asynchronous weak-commitment: AWC

El algoritmo Asynchronous weak-commitment (AWC) [115] es una variación del algoritmo Asynchronous Backtracking en el que los agentes van cambiando su prioridad conforme avanza el proceso de búsqueda. Todos los agentes (varibles) tienen un valor de prioridad, el cual, originalmente, es cero. Los agentes con un valor mayor de prioridad tiene más prioridad, y ante un mismo valor se ordenan alfanuméricamente. Cuando un agente recibe un mensaje *ok?* actualiza su *vista de agente* con los valores del resto de agentes. Si el estado actual del agente satisface todas las restricciones que mantiene con los agentes de mayor prioridad, se dice que el agente mantiene un estado consistente con la *vista de agente*. Si el estado actual del agente no es consistente con la *vista de agente*, se debe seleccionar un nuevo valor el cual sea consistente con la *vista de agente*. Si no existe ningún valor consistente, el valor de prioridad del agente es cambiado a $max + 1$, donde max es el valor de prioridad más alto de todos los agentes con los que está relacionado. Entonces se envía un mensaje *Nogood* a todos los agentes relevantes con el conjunto de valores de las variables que no pueden formar parte de la solución final (esto es necesario para garantizar la completitud del algoritmo). Esta es una de las principales desventajas del algoritmo, ya que para garantizar completitud el agente debe ser capaz de almacenar un número exponencial de mensajes *Nogood*.

Asynchronous Forward-Checking: AFC

El algoritmo Asynchronous Forward-Checking (AFC) es presentado por Amnon Meisels y Roi Zivan en [75]. AFC es un algoritmo de búsqueda distribuida donde los agentes asignan valores a sus variables secuencialmente pero realizan forward checking de manera asíncrona.

El estado del proceso de búsqueda se representa mediante una estructura llamada *Current Partial Assignment* (CPA), la cual comienza vacía y es inicializada por el agente que hace la primera instanciación de variables. Cada

agente, cuando encuentra una instanciación consistente de sus variables la almacena en la CPA y la envía al siguiente agente en un mensaje *CPA*. Cuando un agente no encuentra una instanciación válida, realiza backtrack. Cuando un agente actualiza la CPA, envía una copia de la CPA actualizada, mediante un mensaje *FC_CPA*, a todos los agentes sin asignación para que realicen forward-checking; entonces los agentes concurrentemente filtran sus dominios; si alguno deja su dominio vacío, entonces envía un mensaje *Not_OK* a todos los agentes con las variables no asignadas. Cuando un agente que recibe un mensaje *CPA* ha recibido también un mensaje *Not_OK*, entonces realiza backtrack, asegurando así que solamente se inicializa un backtrack aun cuando se envíen numerosos mensajes *Not_OK*.

Los autores muestran que el algoritmo AFC tiene mejor comportamiento que ABT. Además AFC tiene la ventaja de poder utilizar heurísticas de ordenación ya que realiza la asignación de manera secuencial. La selección de una buena heurística proporciona una gran mejora en el rendimiento del algoritmo.

Asynchronous search with Aggregations: AAS

En [104] M.C. Silaghi, D. Sam-Haroud y B. Faltings presentan el algoritmo Asynchronous search with Aggregations. Este algoritmo tiene la peculiaridad que la distribución se realiza por restricciones en lugar de variables. Así cada agente tiene asignado un conjunto de restricciones y los agentes comparten las variables que forman parte de sus restricciones. Este algoritmo es una extensión del algoritmo ABT donde las restricciones pueden ser datos privados y varios agentes pueden proponer simultáneamente instanciaciones para la misma variable compartida. Cada agente propaga tuplas de valores agregados en lugar de valores individuales. Estas tuplas representan el producto cartesiano de la asignación de diferentes variables. Así, los mensajes intercambiados entre los agentes contienen una lista de dominios (uno por cada variable involucrada), cuyo producto cartesiano representa un conjunto de posibles *soluciones_parciales* al sub-problema del agente que envía el mensaje.

Este algoritmo es particularmente útil cuando existen requerimientos de privacidad en las restricciones del problema, por ejemplo, en los problemas de negociación donde las variables suelen ser públicas pero las restricciones privadas. También resulta útil para problemas con dominios muy grandes ya que la representación de las soluciones mediante el producto cartesiano representa un ahorro importante de espacio. Los autores muestran que en problemas generados aleatoriamente se reduce el número de mensajes intercambiados.

El algoritmo *Maintaining Hierarchical Distributed Consistency* (MHDC) es una extensión del algoritmo AAS en la que se mantiene arco-consistencia durante el proceso de búsqueda distribuida llevada a cabo por AAS. Una de sus principales características es considerar el mantenimiento de consistencia como una tarea jerárquica. De este modo, cumplir la jerarquía de consistencia y llevar a cabo la búsqueda puede ser realizado con alto grado de asincronismo. Esto da al agente mayor grado de libertad y flexibilidad ya que puede contribuir a la búsqueda e incrementar el paralelismo. Los resultados mostrados por las autores indican que se pueden obtener ganancias a nivel computacional al introducir algoritmos de consistencia local distribuida en los algoritmos de búsqueda distribuida.

3.2.2. Algoritmos distribuidos de búsqueda local

Distributed Break-Out: DisBO

En [118] M. Yokoo y K. Hirayama presentan el algoritmo Distributed Break-out (DisBO), el cual se basa en el algoritmo centralizado Break-out [81], algoritmo de búsqueda local con un método para escapar de mínimos-locales. En DisBO cada agente intenta reducir el número de restricciones violadas en las que él está involucrado enviando, a sus agentes vecinos, su actual asignación (mensaje *ok?*) y el valor de su posible mejora (mensaje *improve*). De manera que sólo el agente que proporciona la mayor mejora será el que tiene el derecho a cambiar su valor. Sin embargo, si dos agentes no son vecinos, es posible que

ambos cambien su valor concurrentemente.

En lugar de detectar globalmente que los agentes han caído en un mínimo-local, cada agente detecta si está atrapado en un casi-mínimo-local, lo cual es una condición más débil que un mínimo-local y puede ser detectada mediante comunicaciones locales, y cambia los pesos de las restricciones violadas para salir de él. El agente i está en un casi-mínimo-local cuando viola una restricción y su posible mejora y la de todos sus agentes vecinos es 0. La evaluación de los autores muestra que este algoritmo tiene un peor comportamiento con instancias del problema de coloreado de grafos en las que los problemas son densos o poco densos, ya que estos problemas son relativamente fáciles y no compensa la sobrecarga realizada por DisBO al controlar las acciones concurrentes entre los agentes vecinos. Sin embargo para instancias del mismo problema de gran dificultad el algoritmo DisBO es más eficiente que AWC, cosa que no sucede entre las versiones centralizadas de estos dos algoritmos.

Este algoritmo tiene la desventaja de que a veces no es capaz de resolver situaciones muy simples. Tal y como se describe en [122], ya que el algoritmo entra en un ciclo infinito en el que se van incrementando continuamente los pesos de las mismas restricciones.

Distributed Stochastic Search Algorithm: DSA

Zhang, Wang y Wittenburg presentan en [121] un estudio del algoritmo Distributed Stochastic Search (DSA) [38] aplicado en la resolución de problemas de satisfacción/optimización de restricciones.

El protocolo de ejecución síncrona del algoritmo DSA comienza con un paso de inicialización en el que todos los agentes seleccionan, aleatoriamente, un valor para sus variables. A partir de aquí, los agentes síncronamente van repitiendo los mismos pasos hasta que encuentran la condición de terminación. En cada paso, el agente manda a sus vecinos el valor actualizado de su variable y recibe los valores actualizados de las variables de sus vecinos. Entonces el agente decide si mantiene o cambia el valor de su variable. El proceso de

decisión suele ser estocástico y se basa en su estado actual y en la creencia de los estados de sus vecinos. El objetivo final del proceso de decisión es reducir el número de restricciones violadas. Diferentes estrategias de decisión conducen a diferentes versiones del algoritmo DSA.

Los autores comparan el algoritmo DSA con el algoritmo DisBO y concluyen que una de las versiones de DSA, llamada DSA-B, la cual ejecuta un mayor número de acciones en paralelo que la versión estándar, tiene un mejor comportamiento que el algoritmo DisBO, sobretodo en problemas sobre-restringidos, donde el algoritmo encuentra mejores soluciones y tiene un menor coste de comunicación.

3.3. Algoritmos para CSP distribuidos con complejos problemas locales

En esta sección presentamos los principales algoritmos para resolver problemas de satisfacción de restricciones distribuidos en los que cada agente posee varias variables. La mayoría de ellos son versiones revisadas de los principales algoritmos para DCSP en los que cada agente maneja una única variable. Estas nuevas versiones de los algoritmos se suelen basar en alguno de los dos métodos genéricos de transformación de algoritmos presentados en el capítulo de introducción: *Recompilación de soluciones_parciales* y *Descomposición de sub-problemas*.

3.3.1. Priorización dinámica de Agentes Complejos

A. Armstrong y E. Durfee [13] introducen la prioridad entre agentes para manejar agentes con multiples variables. En este algoritmo, cada agente intenta encontrar una *solución_parcial* a su sub-problema que sea consistente con las *soluciones_parciales* de todos los agentes más prioritarios. Si no existe *solución_parcial*, se produce backtracking o cambio en la prioridad de los

agentes. Además, se examinan varias heurísticas para determinar la mejor ordenación entre los agentes. Una limitación de esta aproximación radica en que si los agentes de mayor prioridad seleccionan una mala solución_parcial (es decir, una solución_parcial que no puede formar parte de la solución global), un agente de menor prioridad tiene que realizar una exhaustiva búsqueda en su problema local para determinar la mala decisión tomada por los agentes más prioritarios. Cuando el problema local es muy grande y complejo, realizar la exhaustiva búsqueda local termina siendo inviable.

Esta aproximación es muy similar al Método 1 propuesto en el apartado 1.4 del capítulo de Introducción. La diferencia entre ambos es que en este método cada agente solo realiza la búsqueda en su sub-problema en el momento requerido, en lugar de tener que buscar todas las soluciones por adelantado.

3.3.2. Versión revisada de AWC

En [70], M. Yokoo y K. Hirayama presentan un algoritmo llamado multi-AWC, en el que cada agente puede manejar multiples variables. El algoritmo está basado en el algoritmo de búsqueda distribuida asynchronous weak-commitment. En este algoritmo cada agente realiza secuencialmente la asignación de cada una de sus variables y se comunica con el resto de agentes cuando encuentra una *solución_parcial* que satisface todas sus intra-restricciones. Más específicamente, cada agente selecciona secuencialmente una variable X_k que es la variable con mayor prioridad entre todas las variables que violan una restricción con variables de mayor prioridad, y modifica el valor de esta variable de manera que no se viole ninguna restricción con variables más prioritarias; si no existe tal valor, el agente incrementa el valor de prioridad de la variable X_k. Cuando todas las variables de un agente satisfacen todas las restricciones con variables de mayor prioridad, el agente envía sus cambios a los agentes con los que está relacionado. En este mensaje se incluye también el valor de la prioridad de cada variable. De esta manera, si un agente de mayor prioridad selecciona una mala *solución_parcial*, el agente de menor prioridad no tiene

que realizar una exhaustiva búsqueda local, sino que simplemente aumenta la prioridad de las variables que violan las restricciones relacionadas con la *solución_parcial* no-válida.

Este algoritmo deriva del método estándar de *descomposición de sub-problemas* para trabajar con agentes con múltiples variables; sin embargo se ha demostrado que es más eficiente que la extensión de AWC usando el método de *descomposición de sub-problemas*.

3.3.3. Versiones revisadas de ABT

K. Hirayama, M. Yokoo y K.P. Sycara evalúan en [51] el esfuerzo computacional y la carga de la red necesaria para resolver problemas del tipo 3-coloring con diferente número de intra-restricciones e inter-restricciones. El algoritmo usado es una extensión de ABT, el cual minimiza el paso de mensajes evitando aquellos que son trivialmente innecesarios; sin embargo, no tiene en cuenta la cuestión referente a la complejidad de los problemas locales.

En [72], A. Maestre y C. Bessiere proponen varias heurísticas para la selección y generación de *Nogoods* y para la mejora en la cooperación entre los agentes, todas ellas aplicadas en una adaptación del algoritmo ABT para múltiples variables. El criterio de almacenaje de *Nogoods* se basa en almacenar solamente un *Nogood* por valor y seleccionar, de entre todos los posibles, aquel cuya variable de menor orden tenga el mayor orden posible; además proponen un algoritmo para la generación de *Nogoods* de tamaño mínimo. Estos criterios consiguen disminuir ligeramente el paso de mensajes pero, por contra, aumentan en gran medida el esfuerzo computacional. En cuanto a la mejora en la cooperación entre agentes proponen una heurística para la ordenación de variables de manera que estarán situadas primero aquellas variables relacionadas con agentes de menor prioridad. De este modo pretenden mejorar la estabilidad de las soluciones, manteniendo una solución lo más parecida posible a la anterior respecto a las variables relacionadas con otros agentes menos

prioritarios. Además proponen una heurística para la selección de valores aplicada únicamente a estas variables (variables relacionadas con agentes de menor prioridad) de manera que se seleccionen los valores que menos reducen el dominio de las variables que todavía no están instanciadas, sin tener en cuenta las variables locales al sub-problema. Esto implica tener un conocimiento del dominio de las variables de otros agentes, lo que implica una gran pérdida de privacidad. Estas últimas heurísticas también reducen el paso de mensajes pero también producen un incremento en el chequeo de restricciones (esfuerzo computacional) cuando se trata de complejos sub-problemas locales. Los autores muestran como la aplicación de todas las heurísticas juntas sobre un conjunto de problemas aleatorios reduce entorno a un 15 % el paso de mensajes pero duplica e incluso triplica en ocasiones (para sub-problemas complejos) el esfuerzo computacional.

3.3.4. Versión revisada de Distributed Break-out

K. Hirayama y M. Yokoo extienden en [50] el algoritmo Distributed Break-out para manejar múltiples variables por agente (multi-DB). Se demuestra que Multi-DB usando *reinicios* y permitiendo que los agentes simultáneamente intercambien las variables que son mutuamente exclusivas obtiene mejores resultado que multi-AWC en problemas 3-SAT.

3.3.5. Versión revisada del método de Recompilación de *soluciones_parciales*

En [24], D.A. Burke y K.N. Brown revisan uno de los métodos estándar para la generalización de algoritmos de resolución de DCSP con multiples variables por agentes. Exactamente se basan en el método de *Recompilación de soluciones_parciales* e introducen los conceptos de *dominación* e *intercambiabilidad* para reducir la búsqueda en los sub-problemas. En cuanto a la reformulación

del problema mediante la recompilación de *soluciones_parciales*, proponen buscar únicamente una *solución_parcial* para cada una de las combinaciones de las variables externas (variables relacionadas con otros agentes). De este modo el espacio de *solucinones_parciales* se ve reducido en un factor d^{np}, donde d es la talla del dominio, n el número de agentes y p el número de variables privadas (variables no relacionadas con otros agentes), reduciendo así tanto el tiempo de cómputo de las *soluciones_parciales* de cada sub-problema como el almacenamiento de las mismas. También proponen, tras la reformulación del problema, identificar los valores que son intercambiables con respecto a un subconjunto de agentes vecinos. Esto implica que cada agente conozca las relaciones existentes entre el resto de agentes, lo cual no suele suceder en la mayoría de algoritmos para la resolución de DCSPs. Además si el número de relaciones entre agentes es elevado, la reducción en los dominios del problema reformulado es mínima. Estas técnicas son aplicables tanto en DCSP como DCOP aunque parecen ser más relevantes para DCOP. De cualquier modo, su aplicabilidad dependerá del número de variables externas de los sub-problemas, pues si este número sigue siendo muy grande, el cálculo y almacenamiento de todas las *soluciones_parciales* puede seguir siendo inviable.

3.4. Algoritmos distribuidos para problemas de optimización de restricciones

Los DCOP son utilizados para la resolución de problemas de optimización que son por naturaleza distribuidos. En este apartado presentamos los principales algoritmos para la resolución de problemas de optimización de restricciones distribuidos: DPOP y ODPOP.

3.4.1. Distributed pseudotree optimization procedure for general networks: DPOP

A continuación describimos el algoritmo DPOP [84] presentado por Adrian Pectu y Boi Faltings.

El primer paso para la ejecución del algoritmo es convertir el grafo de restricciones en un *pseudotree*. Un grafo con disposición de *pseudotree* es un árbol con un vértice raíz y con los mismos vértices que el grafo original, pero con la propiedad que todos los vértices adyacentes del grafo original están en la misma rama del árbol.

El segundo paso es la fase de propagación *UTIL*. La propagación comienza en las hojas del *pseudotree* y sube hacia arriba (es enviado a los padres). El mensaje *UTIL* lleva información de la utilidad que puede alcanzar el nodo hijo con respecto a todos los valores posibles del padre. En una red con ciclos, la utilidad de un hijo puede depender además de los valores de las variables anteriores al padre (variables del contexto). Entonces el mensaje *UTIL* lleva la información de la utilidad para cada una de las posibles combinaciones de los valores de todas las variables (variables del contexto) que pueden afectar al nodo hijo que envía el mensaje (son mensajes multidimensionales).

Cuando el nodo raíz recibe los mensajes *UTIL* de todos sus hijos, ya es capaz de calcular la utilidad óptima para cada uno de sus valores, entonces escoge el valor óptimo y comienza la propagación *VALUE*.

El tercer paso es la fase de propagación *VALUE*, en esta fase, cada padre comunica a sus hijos la asignación de valores escogida por él y por las variables del contexto, por lo que cada hijo es capaz de calcular su asignación óptima y pasarla a sus hijos; el proceso continua hasta alcanzar todos los nodos hoja.

El número de mensajes que produce el algoritmo es líneal:

- $n - 1$ mensajes *UTIL* (n es el número de nodos).
- m mensajes *VALUE* (m es el número de ejes).
- $2 * m$ mensajes para construir el *pseudotree*.

La complejidad del algoritmo recae en el tamaño del mensaje *UTIL* que es exponencial con respecto a la anchura inducida del *pseudotree*. La anchura inducida corresponde al máximo número de padres de cualquier nodo del grafo inducido. Siempre se produce el peor caso de complejidad.

3.4.2. Open/Distributed Constraint Optimization: ODPOP

El algoritmo ODPOP [85] es una revisión del algoritmo DPOP presentado por los mismos autores: Adrian Pectu y Boi Faltings.

ODPOP toma ventaja de los algoritmos de búsqueda ya que no siempre cae en el peor caso de complejidad y produce mensajes de tamaño lineal. También toma ventaja de los algoritmos de programación dinámica ya que produce pocos mensajes. Es aplicable a problemas de optimización de restricciones abiertos, es decir, con dominios no acotados.

El algoritmo ODPOP, al igual que DPOP, tiene tres pasos. El primero, la construcción del *pseudotree* es igual que en DPOP. El segundo es la fase *ASK/GOOD* que difiere del segundo paso de DPOP (mensajes *UTIL*). El tercer paso, mensajes *VALUE*, es igual que en el algoritmo DPOP.

Durante la segunda fase *ASK/GOOD* los padres piden a los hijos sugerencias de valores para sus variables y los hijos responden con mensajes *GOOD* que contienen una única asignación de la variable del padre y de las variables contexto del padre y la utilidad que esa asignación produce (el tamaño del mensaje es lineal). Cada nodo siempre envía los mensajes *GOOD* en orden de utilidad decreciente por lo que un nodo padre no necesita esperar a que todos sus hijos le envíen sus mensajes para determinar cual es la asignación óptima (no siempre cae en el peor caso). Cuando el nodo raíz determina que ya ha recibido los suficientes mensajes de sus hijos para determinar la asignación óptima, comienza la fase *VALUE*.

El número de mensajes y la memoria requerida por ODPOP es d^w donde d es la talla máxima del dominio y w el ancho inducido del grafo de restricciones.

3.5. Conclusiones

A lo largo del capítulo hemos realizado una revisión del estado del arte de los principales algoritmos para la resolución de problemas de satisfacción de restricciones distribuidos. La mayoría de los algoritmos desarrollados por la comunidad científica están pensados para que cada *"agente"* asigne valores a una única variable. Posteriormente, muchos de ellos han sido adaptados para que cada *"agente"* pueda manejar varias variables. Sin embargo estas adaptaciones no suelen ser aplicables a problemas de gran complejidad debido a sus excesivos requerimientos espaciales o computacionales, ya que o bien almacenan un gran número de soluciones_parciales y/o mensajes, o bien realizan muchos intercambios de mensajes.

Capítulo 4

Resolución de CSPs mediante particionamiento

Normalmente, los problemas de satisfacción de restricciones que modelan problemas reales implican un gran número de variables y restricciones. Esta clase de problemas puede ser manejada de manera global solo a costa de un gran esfuerzo computacional. Por esto, podría ser una gran ventaja particionar esta clase de problemas en un conjunto de sub-problemas más simples, de tal modo que cada sub-problema pueda ser resuelto más fácilmente. Además, puede ser aconsejable (no necesario) particionar un problema por las características del mismo. En general el particionamiento de un problema es bueno en lo que se refiere a eficiencia debido a que los sub-problemas resultantes pueden ser manejados de manera más sencilla. En este capítulo se proponen tres métodos para el particionamiento de problemas: el primero de ellos está basado en el particionamiento de grafos; el segundo realiza una búsqueda de estructuras de árbol dentro del problemas; y el tercero trata de identificar entidades características del problema en estudio.

4.1. Notación

A continuación se presentan algunas definiciones necesarias para la comprensión de los métodos de descomposición de CSPs que se proponen en los siguientes apartados.

Definición 4.1.1. Un **grafo de restricciones** $G = (V, A)$ se compone de un conjunto de nodos V =$\{v_1$,v_2,...,$v_n\}$, y un conjunto de pares de nodos, llamados aristas, A =$\{\{v_1,\ v_2\}$,$\{v_1,\ v_3\}$,...$\}$. Un CSP binario puede representarse gráficamente mediante un *grafo de restricciones*, de tal manera que cada variable x_i del CSP es representada por un nodo v_i en el *grafo de restricciones*, y cada restricción del CSP entre dos variables x_i, x_j es representada por una arista $\{v_i, v_j\}$ en el *grafo de restricciones*.

Definición 4.1.2. Un grafo $G' = (V', A')$ es un **sub-grafo de restricciones** de un *grafo de restricciones* $G = (V, A)$ si $V' \subseteq V$ y $\forall v_i, v_j \in V' \{v_i, v_j\} \in A' \rightarrow \{v_i, v_j\} \in A$.

Definición 4.1.3. Dado un *grafo de restricciones* G, un **particionamiento** $D(G)$ **del grafo G** es un conjunto de sub-grafos $\mathcal{G} = \{G'_1, G'_2, ..., G'_k\}$ donde cada $G'_i = \{V'_i, A'_i\}$ es un sub-grafo de G y $V'_1 \bigcup V'_2 \bigcup ... \bigcup V'_k = V$ y $\forall\ i,\ j$, $i \neq j$: $V_i \bigcap V_j = \emptyset$.

4.2. Partición y descomposición

En general cuando se habla de división de grafos, lo que se pretende es seleccionar un conjunto de nodos o aristas, cuya eliminación transforma el grafo original en un conjunto disjunto de sub-grafos cuya cardinalidad sigue alguna relación deseada. La mayoría de las veces, el interés recae en la cardinalidad del conjunto de nodos o aristas suprimidos, aunque en algunos trabajos minoritarios también ha sido interesante estudiar la estructura teórica del sub-grafo inducido por la supresión del conjunto de nodos o aristas y así aprovechar las ventajas del tipo de grafo inducido, por ejemplo, en [78] se busca que el sub-grafo inducido sea hamiltoniano.

Dentro del campo de la división de grafos es importante diferenciar entre la *partición* y la *descomposición* de grafos. Cuando se quiere dividir un grafo en k sub-grafos, lo que se lleva a cabo es una *partición* del grafo. Por el contrario se habla del árbol de *descomposición* de un grafo, cuando el grafo original se quiere trasformar en un árbol cuyos nodos son sub-grafos del grafo original de tal manera que el nodo raíz es el grafo original y, recursivamente, los hijos de los nodos del árbol son los sub-grafos en los que se descompone dicho nodo.

En este capítulo presentamos tres técnicas para el particionamiento de grafos. La primera de ellas busca seleccionar un conjunto de aristas de tal manera que los sub-grafos resultantes del particionamiento tengan un tamaño similar y estén mínimamente conectados. La segunda técnica de particionamiento se caracteriza por la estructura de los sub-grafos resultantes del particionamiento, los cuales tienen la peculiaridad de ser árboles. Finalmente, la tercera técnica de particionamiento se basa en estudiar las características del problema e identificar entidades propias de cada problema, de manera que cada entidad puede ser representada como un sub-grafo.

4.3. Clasificación de CSPs

Existen muchas maneras para resolver un CSP. Sin embargo, un paso importante antes de seleccionar el método de resolución es identificar el tipo de problema con el que estamos trabajando. Nosotros clasificamos los tipos de problemas en tres categorías: Problemas Centralizados, Problemas Distribuidos y Problemas Particionables:

1. Se dice que un CSP modela un problema centralizado cuando no existen reglas de privacidad o seguridad entre diferentes partes del problema y todo el conocimiento sobre el problema puede ser canalizado en un único proceso. Un CSP centralizado es aconsejables ser resuelto mediante un resolvedor de CSPs centralizado. Muchos problemas son reconocidos como típicos ejemplos para ser modelados como CSPs centralizados y resueltos mediante técnicas de programación de restricciones. Algunos ejemplos son: sudoku, k-reinas, coloreado de grafos, etc.

2. Se dice que un CSP modela un problema distribuido cuando las variables, dominios y restricciones de la red subyacente están distribuidos entre agentes. Esta distribución se puede deber a diferentes causas: por motivos de privacidad en el problema, donde se identifican entidades las cuales agrupan a un conjunto de variables y restricciones, y cada entidad trabajaría con información estratégica que no puede ser revelada

a los competidores, ni incluso a una autoridad central; por cuestiones de seguridad, donde se identifican agentes de modo que aunque uno de ellos fallase los demás podrían ser capaces de encontrar la solución al problema. Ejemplos de este tipo de problemas son las redes de sensores, meeting scheduling, aplicaciones de la red Internet, etc. Este tipo de problemas debe ser resuelto de manera distribuida.

3. Se dice que un CSP es particionable cuando modela un problema que puede ser dividido en problemas más pequeños (sub-problemas) relacionados entre sí, los cuales se resolverán mediante reglas de coordinación. Por ejemplo, el espacio de búsqueda de un CSP puede ser descompuesto en varias regiones de modo que la solución del problema se podría encontrar usando técnicas de computación paralela. Este tipo de problemas es aconsejable resolverlos de manera distribuida.

Es importante hacer notar que un problema es centralizado o distribuido por naturaleza, lo cual es una característica inherente del problema. Por lo tanto, un CSP distribuido no puede ser resuelto mediante técnicas centralizadas. Sin embargo, ¿un CSP inherentemente centralizado podría ser resuelto mediante técnicas distribuidas? La respuesta es "SI"si nosotros somos capaces de particionar el CSP.

La mayoría de problemas de satisfacción de restricciones que modelan situaciones de la vida real terminan convirtiéndose en extensas redes de variables interconectadas. Manejar toda la red de manera centralizada puede resultar inviable debido a su coste espacial y/o computacional. Una buena opción para resolver esta situación es hacer uso del paradigma "*divide y vencerás*", es decir, particionar el problema de manera que obtengamos un conjunto de subproblemas que debido a su tamaño más reducido son más fácilmente resolubles.

A continuación podemos comprobar, con un ejemplo, como un CSP centralizado podría ser particionado en varios sub-problemas para obtener un conjunto de sub-CSPs más simples. De esta manera, podremos aplicar técnicas distribuidas para resolver el CSP particionado.

El problema de coloreado de mapas es un típico problema centralizado. Como ya comentamos en capítulos anteriores, el objetivo de este problema es colorear un mapa de tal manera que regiones adyacentes tengan diferentes colores. Imaginemos que tenemos que colorear los países europeos (Figura 4.1(2)). En la Figura 4.1(1) podemos ver una porción de Europa ya coloreada. La solución a este problema puede encontrarse usando un resolvedor de CSPs centralizado. Sin embargo, si el problema fuese colorear todas las regiones de cada país europeo (España, Figura 4.1(3); Francia, Figura 4.1(4)), parece claro que el problema podría ser particionado en un conjunto de sub-problemas en el que cada sub-problema representa el problema de colorear las regiones de cada país. De este modo, el problema de coloreado de mapas podría ser resuelto con técnicas distribuidas aún cuando el problema no sea inherentemente distribuido.

A continuación se proponen tres métodos para el particionamiento de CSPs: el primero de ellos se basa en la identificación de clusters usando técnicas de particionamiento de grafos; el segundo consiste en la identificación de estructuras de árbol; y el tercero realiza un estudio en profundidad del problema en cuestión para poder identificar entidades propias del mismo.

4.4. Identificación de clusters

La primera técnica que proponemos para el particionamiento de un CSP se basa en el uso de técnicas generales para el particionamiento de grafos. Muchos investigadores están trabajando en el particionamiento de grafos [49], [2]. El principal objetivo del particionamiento de grafos es dividir un grafo en un conjunto de sub-grafos de tal manera que cada sub-grafo tenga aproximadamente el mismo número de nodos y la suma de todas las aristas conectando diferentes regiones sea minimizada. Ambas características son beneficiosas para los sub-problemas resultantes del particionamiento de un CSP. De este modo, al tener sub-problemas con aproximadamente el mismo número de nodos, conseguimos obtener sub-problemas con un coste computacional equilibrado. Además, al

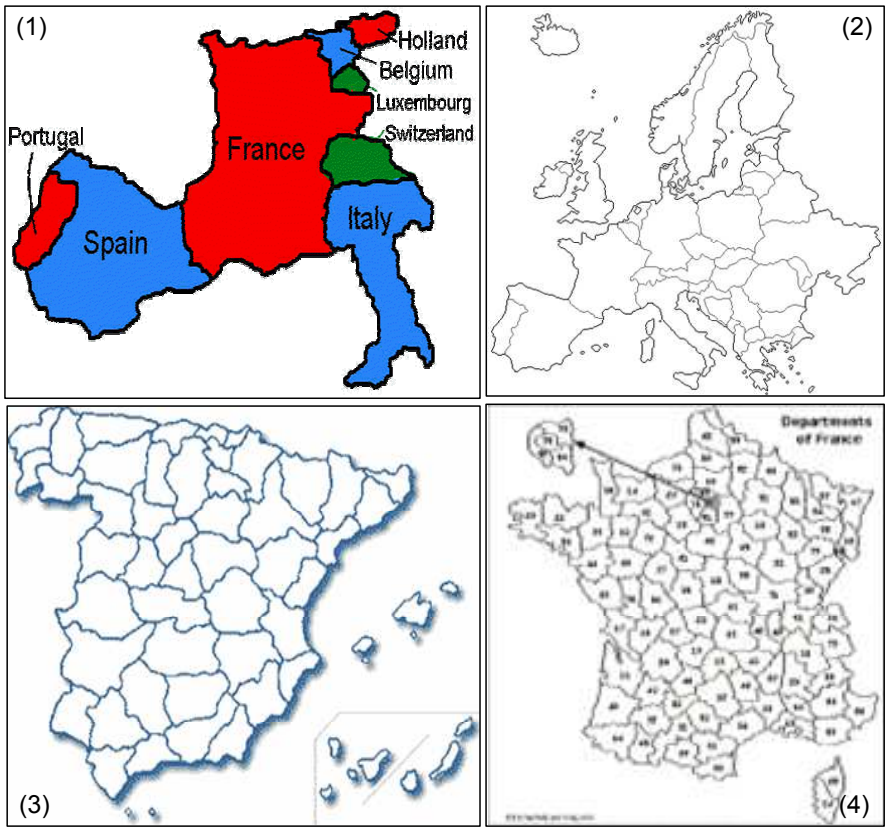


Figura 4.1: Coloreado del mapa de Europa.

minimizar las aristas de conexión entre los diferentes sub-problemas, conseguimos que el proceso de coordinación entre ellos sea más simple y se minimice la comunicación necesaria para encontrar una solución global del problema.

Este tipo de particionamiento (mínimas conexiones y particiones equilibradas) no es una tarea fácil, sin embargo existen muchas heurísticas que consiguen resolver este problema de una manera muy eficiente. Por ejemplo, grafos con 14000 nodos y 410000 aristas pueden ser particionados en menos de 2 segundos [60]. Por lo tanto, aplicar técnicas de particionamiento de grafos puede ser una buena idea para particionar un CSP en semi-independientes sub-CSPs.

Específicamente, en los trabajos realizados en esta tesis ([95], [8], [5], [7],

[12], [106], [98]), el particionamiento de los CSPs, usando técnicas de particionamiento de grafos, se realiza mediante el software de particionamiento de grafos llamado METIS [60]. Los algoritmos utilizados por METIS ([61], [62]) se basan en técnicas de particionamiento de grafos multi-nivel gracias a las cuales consigue particiones de gran calidad de manera muy rápida. METIS proporciona dos programas *pmetis* y *kmetis* para llevar a cabo el particionamiento de grafos no estructurados. Este software divide el grafo en k partes aproximadamente iguales. El número de particiones requeridas debe ser indicado por el usuario. Esto nos aporta la libertad de poder probar para un mismo problema particiones de diferentes tamaños. Sin embargo, no es fácil determinar, a priori, el tamaño de la mejor partición.

Usando METIS, particionamos un CSP en varios sub-CSPs de manera que el número de variables de cada sub-CSP sea similar y además se minimize el número de restricciones entre variables de diferentes sub-CSPs.

En la Figura 4.2 podemos ver un ejemplo de una red de restricciones formada por 9 nodos (variables) y 14 aristas (restricciones). También observamos el particionamiento llevado a cabo por el software METIS cuando se le solicitan tres particiones. De este modo conseguimos tres clusters con un número de nodos balanceado (3 nodos por cluster) y mínimamente interconectados.

4.5. Identificación de árboles

Como Rina Dechter dice en [30], un problema es considerado fácil cuando podemos encontrar una solución en tiempo polinomial. En el contexto de CSPs, un problema se considera fácil si un algoritmo de búsqueda puede resolver el problema sin retroceder, es decir, sin realizar ninguna vuelta atrás (bactrack); encontrando, de este modo, una solución en tiempo lineal, dependiente del número de variables y restricciones. Las investigaciones realizadas en este campo (ver [28]) han identificado característica topológicas que determinan este nivel de consistencia y se han creado algoritmos capaces de transformar algunas redes en representaciones libres de backtrack.

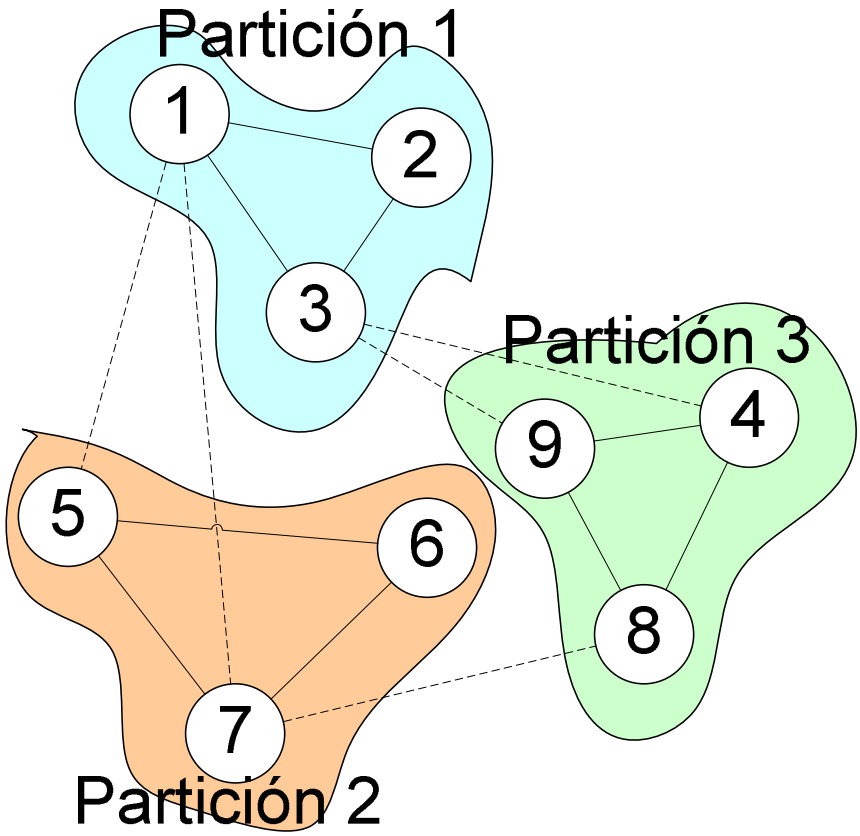


Figura 4.2: Ejemplo de particionamiento de grafos llevado a cabo por el software METIS.

Dichas investigaciones se centran en un parámetro gráfico llamado *anchura*. Las definiciones son relativas al grafo de restricciones primario. Un grafo de restricciones (primario) ordenado se define como un grafo donde los nodos están linealmente ordenados representando la secuencia de asignaciones ejecutadas por el algoritmo de backtracking. La *anchura* de un nodo es el número de arcos que conectan ese nodo con los previos. La *anchura* de una ordenación es la máxima de las *anchuras* de todos los nodos y la *anchura* de un grafo es la mínima anchura de todas las ordenaciones del grafo. Un grafo de restricciones ordenado está libre de backtracks si el nivel de consistencia local direccional a lo largo de este orden es mayor que la *anchura* del grafo ordenado. De este

modo, si el grafo tiene *anchura uno*, sólo se requiere una arco-consistencia direccional [30] para resolverlo en tiempo lineal. Solamente los árboles son grafos con *anchura uno* [39].

Cualquier CSP binario puede ser descompuesto en un conjunto de árboles interconectados. Sin embargo existen muchas maneras para realizar esta descomposición. Dependiendo de los requerimientos del usuario, podría ser deseable que los diferentes árboles obtenidos tuviesen un número de nodos equilibrado; o tal vez, que el número de interconexiones entre los nodos de diferentes árboles sea el mínimo posible.

Las dos características anteriormente comentadas serían deseables para la resolución de un CSP con técnicas distribuidas, ya que con ello conseguiríamos tener sub-problemas equilibrados, con lo que la resolución de cada subproblema tendría un coste similar. Además al minimizar el número de aristas entre diferentes sub-problemas, en principio sería más sencilla la coordinación de todos los sub-problemas para encontrar la solución global. Sin embargo, encontrar la solución que optimize ambos objetivos tiene una gran complejidad. Investigaciones en este campo han demostrado que prácticamente cualquier criterio no trivial de particionamiento de un grafo es un problema NP-completo [90], recayendo de este modo en un problema de coste computacionalmente intratable. Por este motivo y ya que no es el objetivo central de esta tesis encontrar el mejor particionamiento de árboles, se ha optado por usar una técnica ([6]) que realiza en tiempo polinomial (n^2 en el peor caso, donde n es el número de variables) una partición del grafo en árboles no equilibrados (ver Algoritmo 1). Esta técnica particiona el problema en un conjunto de árboles interconectados, donde el número de árboles no es conocido a priori. De esta manera, conseguimos particionar un CSP, que en general suele ser un problema de gran complejidad, en un conjunto de sub-problemas que debido a su estructura de árbol pueden ser resueltos muy fácilmente.

El algoritmo *ParticionarArboles* (Algortimo 1) se encarga de particionar el *grafo de restricciones* G en un conjunto de *sub-grafos*, cada uno de los cuales tiene una estructura de árbol. Para ello, mientras queden nodos en el grafo G

sin pertenecer a ningún árbol, selecciona aleatoriamente un nodo del grafo G que será el nodo raíz del siguiente árbol. A continuación llama a la función *buscarArbol* que se encarga de construir un árbol seleccionando nodos de G que todavía no pertenecen a ningún otro árbol.

```
Algoritmo ParticionarArboles(G)
;Grafo de restricciones G, originalmente ningún nodo está seleccionado.
begin
    ListaArboles= Ø;
    while(G≠ Ø)
        Arbol= Ø;
        NodoRaiz=selecNodo(G);          ;escoge un nodo no seleccionado
        insertarNodo(NodoRaiz,Arbol);
        marcar NodoRaiz como seleccionado;
        buscarArbol(G,NodoRaiz,Arbol);
        insertarArbol(Arbol,ListaArboles);
end ParticionarArboles

funcion buscarArbol(G,NodoPadre,Arbol)
begin
    for each nodo i ϵ G adyacente a NodoPadre y no seleccionado do
                                    ;i es adyacente a NodoPadre si al menos existe
        if noCiclo(i,Arbol)         ;una restricción entre i y NodoPadre.
            insertarNodo(i,Arbol);
            marcar i como seleccionado;
            buscarArbol(G,i,Arbol);
end buscarArbol
```

Algoritmo 1. Algoritmo para la descomposición de un grafo en árboles.

La función *buscarArbol* hace una búsqueda en profundidad en el grafo G

para construir un sub-grafo con estructura de árbol. Para ello, la función *buscarArbol* selecciona un nuevo nodo i el cual está relacionado con el anterior nodo introducido en el árbol (*NodoPadre*) y que no forma ningún ciclo con los nodos ya introducidos en el actual árbol, es decir, no está relacionada con ninguna otra variable ya seleccionada en este árbol. Este nodo i se marca como *seleccionado* indicando que ya pertenece a un árbol y se llama recursivamente a la función *buscarArbol*. Si un nodo tiene varios nodos adyacentes, será igualmente correcto seleccionarlos en cualquier orden. Sin embargo, es muy importante retrasar la comprobación de si el nodo está o no seleccionado hasta que terminen las llamadas recursivas de los nodos previamente seleccionados. La construcción de un árbol termina cuando ya no quedan variables no seleccionadas o cuando cualquier nueva variable que se introdujese en el árbol crease un ciclo. El proceso de particionamiento del grafo G finaliza cuando todos los nodos de G forman parte de algún árbol.

En la Figura 4.3 podemos ver un ejemplo de una red de restricciones formada por 13 nodos (variables) y 18 aristas (restricciones). También observamos el particionamiento llevado a cabo por el algoritmo *ParticionarArboles* que ha dividido el problema en cuatro sub-problemas con estructura de árbol.

4.6. Particionamiento basado en entidades

En los apartados anteriores hemos visto cómo, de manera general, cualquier CSP puede ser particionado con técnicas que no tienen en cuentan características propias del dominio del problema. Sin embargo, en esta sección proponemos particionar un problema, que en principio es centralizado, mediante la identificación de entidades que son características de dicho problema, de manera que cada entidad represente un sub-problema. Esto requiere un análisis y conocimiento en profundidad del problema que queremos particionar, que puede conducirnos a una más adecuada distribución del problema para ser resuelto de manera más eficiente.

Este método de particionamiento toma como punto de partida considerar la

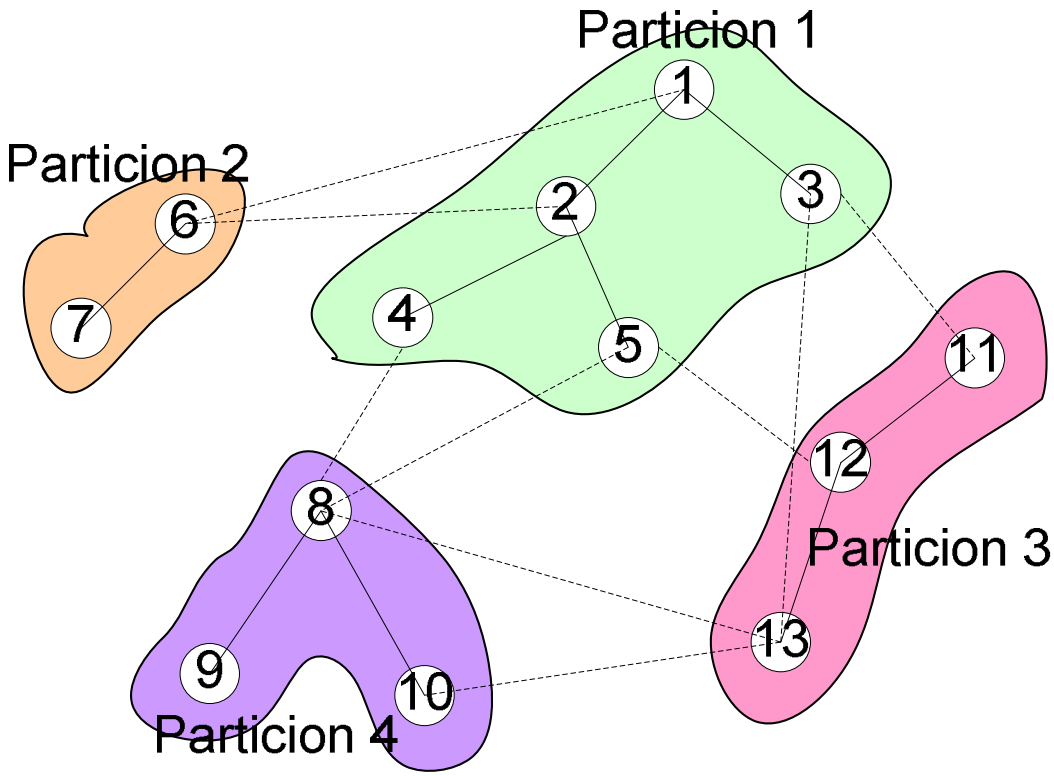


Figura 4.3: Ejemplo de particionamiento de grafos llevado a cabo por el algoritmo *ParticionarArboles*.

existencia de entidades, que representan partes concretas del problema: objetos reales o abstractos, personas, actividades de un sistema, etc, las cuales son propias del problema en estudio. Las entidades con las mismas características forman un tipo de entidad. Cada entidad engloba un conjunto de variables y restricciones que están claramente vinculadas con la entidad. Además, uno de los puntos fuertes de este modelo es que prevé que las entidades puedan mantener relaciones entre ellas, de manera que pueden existir un conjunto de variables o restricciones que modelizan las relaciones entre entidades. El principio fundamental en este tipo de particionamiento es que hechos distintos deben ser manejados por entes distintos.

Para identificar las entidades dentro del sistema deben conocerse a fondo

las características del problema en estudio, el objetivo perseguido con la resolución del problema y el tipo de modelización que se ha realizado del mismo. De este modo podremos identificar qué variables y restricciones están asociadas inequívocamente a cada entidad y establecer las inter-relaciones entre las diferentes entidades, pudiendo incluso crear una jerarquía según su importancia dentro del contexto del problema.

A continuación mostraremos con un ejemplo concreto cómo es posible realizar la identificación de diferentes tipos de entidades para un mismo problema. El problema seleccionado para llevar a cabo este tipo de particionamiento es la *planificación de horarios ferroviarios*.

4.6.1. Particionamiento en la planificación ferroviaria

Debido a la desregularización de los operadores ferroviarios, cada día se planifican más trayectos ferroviarios de grandes distancias. La planificación de estos trayectos involucra el manejo de los datos de un gran número de estaciones ferroviarias situadas en diferentes países, con diferentes políticas ferroviarias. Por ejemplo, para viajar de Londres a Madrid, el tren debe recorrer tres países distintos con diferentes políticas en materias de regularidad, privacidad, prioridades, etc. Cada operador ferroviario mantiene diferentes estrategias operativas y comerciales las cuales pueden tratar con información privada que no quieren que sea revelada a sus competidores. Todas estas razones nos han motivado a pensar que mediante el particionamiento del problema de *planificación de horarios ferroviarios* ganaremos en eficiencia, operatividad y privacidad a la hora de resolver el problema.

Tradicionalmente, la *planificación de horarios ferroviarios* se genera manualmente dibujando el trayecto de cada tren en un diagrama de espacio tiempo, lo que se suele denominar *Malla Ferroviaria* (ver Figura 4.4). El horario de un tren se genera desde un tiempo inicial de salida de la estación inicial y es manualmente ajustado a medida que se va avanzando en el tiempo y el espacio de manera que todas las restricciones se cumplan. Normalmente, los trenes de mayor prioridad son planificados primero, y a continuación los trenes de menor

prioridad. Este proceso puede tardar días, y normalmente termina cuando se encuentra la primera solución factible, aunque esta solución probablemente no será la más óptima en cuanto al tiempo de recorrido de los trenes [97].

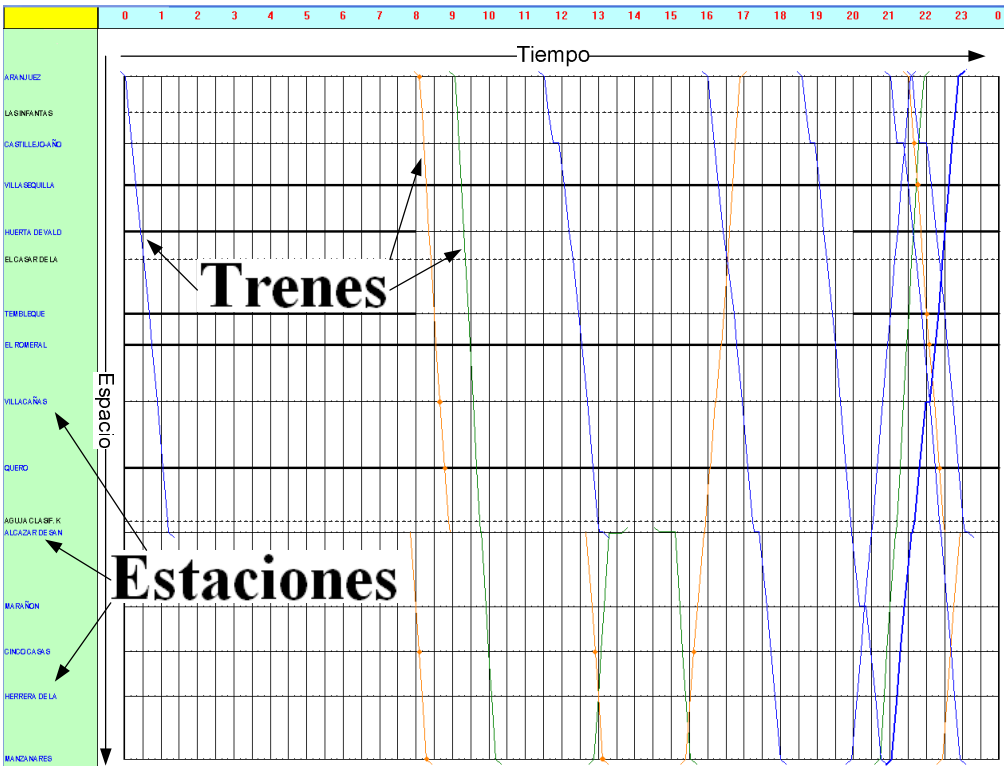


Figura 4.4: Ejemplo de *Malla Ferroviaria*.

En su versión automatizada ([110], [16], [67], [59], [109], [55], [107]), la *planificación de horarios ferroviarios* es un problema cuya resolución requiere un gran coste computacional, particularmente en el caso de redes ferroviarias reales ([57], [68], [20]), por donde circulan un gran número de trenes de características muy dispares y donde el número y la complejidad de las restricciones crece drásticamente. Un horario factible para un tren debe indicarnos los tiempos de salida y de llegada del tren a cada una de las estaciones por las que pasa durante su recorrido. Estos tiempos deben cumplir todas las restricciones operativas impuestas en el problema, tales como: capacidad de las vías, capacidad de las estaciones, tiempos mínimos de seguridad para la gestión de

cruces, tiempos de recorrido de los tramos, tiempos de sucesión entre trenes consecutivos, etc.

La modelización del problema de *planificación de horarios ferroviarios* como un problema de satisfacción de restricciones (CSP) ([19], [96], [69]), requiere de un conjunto de variables con dominios discretos; cada una de ellas representando un instante de llegada o de salida de un tren a una estación. Además habrá un conjunto de restricciones modelizando todos los requisitos que debe cumplir un horario para que sea factible, como pueden ser, los requerimientos de usuarios, las normas del tráfico ferroviario y las limitaciones de la infraestructura ferroviaria.

Después de estudiar las características del problema ([4], [56], [94]), se pueden identificar dos tipos de entidades que juegan un papel principal en el problema: los *trenes* y las *estaciones*. De este modo podríamos plantear dos tipos de particionamiento basados en estas entidades:

- El primer tipo de particionamiento estará basado en la entidad *tren*. De este modo, en su versión más distribuida, el problema se particiona en tantos sub-problemas como trenes hay que planificar. Cada sub-problema engloba las variables correspondientes a los instantes de tiempo de salida/llegada de un único tren en todas las estaciones de su trayecto, así como las restricciones existentes entre estas variables. De este modo la resolución de cada sub-problema consistirá en la asignación del horario de un tren. Dependiendo del número de particiones deseadas, cada sub-problema podría englobar uno o varios trenes. De este modo, otro tipo de partición que se basa en la entidad *tren*, sería englobar en un mismo sub-problema todos los trenes del mismo tipo (regionales, mercancías, cercanías, etc.) o del mismo operador ferroviario, así cada sub-problema se encargaría de asignar los horarios a los trenes de una categoría.

  Este tipo de particionamiento tiene dos importantes ventajas. La primera, nos ayuda a mejorar la privacidad, ya que la información estratégica

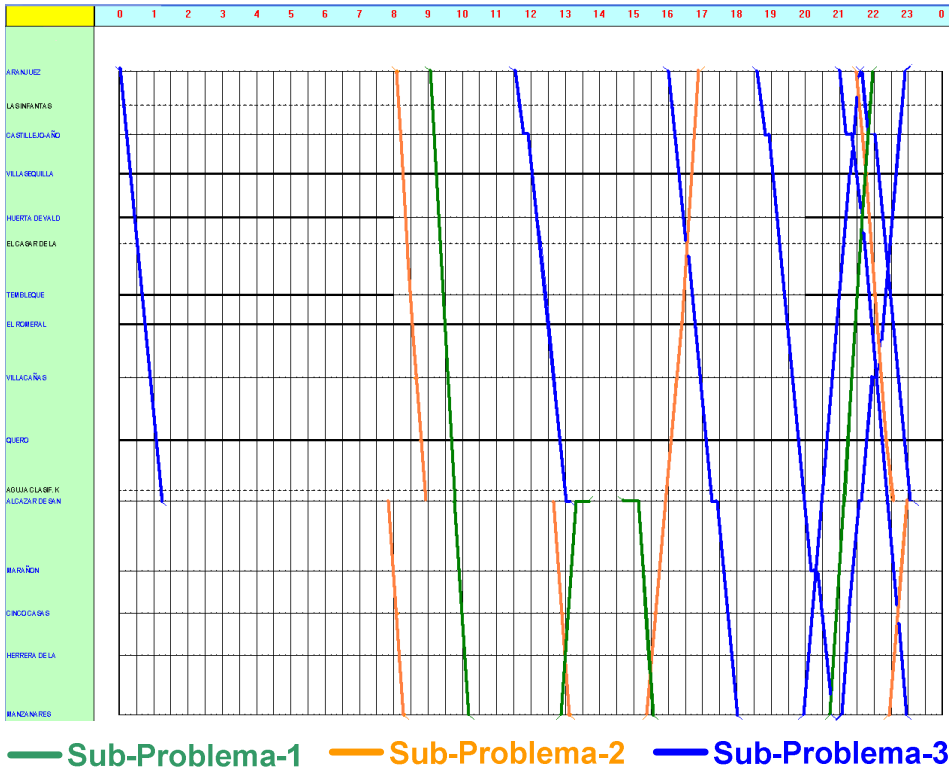


Figura 4.5: Ejemplo de particionamiento basado en la entidad *tren*. Cada sub-problema engloba los trenes del mismo tipo.

de los diferentes operadores ferroviarios, que comparten la misma infraestructura, no tendría que ser revelada a sus competidores, pues las restricciones relacionadas con sus propios trenes sólo necesitan ser mencionadas en los sub-problemas correspondientes a los trenes del operador en cuestión. La segunda ventaja, es que este tipo de particionamiento nos permite manejar eficientemente las prioridades entre trenes de tipos diferentes. Para esto, simplemente será necesario resolver primero los sub-problemas correspondientes a los trenes más prioritarios, los cuales obtendrán mejores tiempos de recorrido al estar menos restringidos por los horarios de trenes previamente planificados.

En la Figura 4.5 podemos ver un ejemplo de particionamiento del problema de *planificación de horarios ferroviarios*. En este ejemplo cada

sub-problema engloba las variables y restricciones correspondientes a todos los trenes del mismo tipo, dando lugar a tres particiones.

- El segundo tipo de particionamiento se basa en la entidad *estación*. De este modo, el particionamiento del problema es llevado a cabo por medio de la selección de grupos de estaciones contiguas, de manera que cada sub-problema engloba las variables correspondientes a los instantes de salida/llegada de todos los trenes a un grupo determinado de estaciones. El trayecto de un tren puede implicar que el tren circule por diferentes regiones o países con diferentes políticas ferroviarias, por lo que parece lógico que este tipo de particionamiento separe en diferentes sub-problemas los tramos de diferentes países, de manera que cada uno de ellos sea gestionado de acuerdo a sus políticas ferroviarias. Además, para llevar a cabo un correcto particionamiento, también sería interesante analizar la infraestructura ferroviaria de cada zona y detectar las zonas más congestionadas, cuya planificación ferroviaria es mucho más complicada.

  Este tipo de particionamiento tiene la ventaja de poder obtener sub-problemas con una carga equilibrada, ya que además de tener en cuenta el número de estaciones, también se puede tener en cuenta la complejidad que cada una de ellas tiene. Así, un sub-problema puede englobar muchas estaciones las cuales son poco restrictivas, mientras que otro sub-problema puede englobar pocas estaciones pero las cuales son cuellos de botella. Otra ventaja de este tipo de particionamiento es que los sub-problemas que contienen estaciones las cuales son cuellos de botella pueden ser resueltos primero para poder encontrar una solución de manera más eficiente.

  Así, una *Malla Ferroviaria* entre dos ciudades es particionada en un conjunto de *Mallas Ferroviarias* más pequeñas las cuales agrupan un bloque de estaciones contiguas. En la Figura 4.6 podemos ver como la *Malla Ferroviaria* de la Figura 4.4 ha sido particionada en 3 sub-problemas.

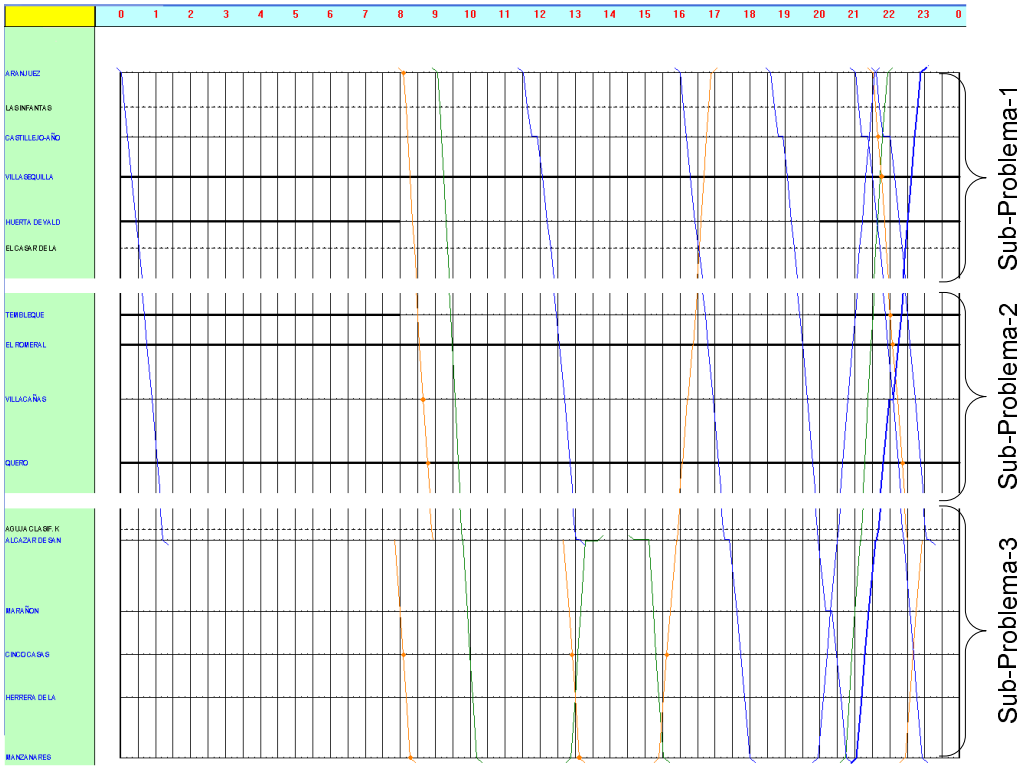


Figura 4.6: Ejemplo de particionamiento basado en la entidad *estación*.

4.7. Conclusiones

En este capítulo se ha realizado una clasificación de los CSPs en tres grupos de problemas: centralizados, distribuidos y particionables. Tras la identificación de cada uno de estos tipos, el capítulo se centra en los CSPs particionables. Nuestra propuesta es resolver un CSP particionable, el cual puede no ser un problema inherentemente distribuido, mediante técnicas distribuidas. Para ello, el primer paso es particionar el problema. Al respecto proponemos tres métodos de particionamiento: la identificación de clusters, la identificación de estructuras de árbol, y la identificación de entidades propias del problema. La calidad del particionamiento realizado por cada uno de estos métodos será evaluado en el Capítulo 7.

Capítulo 5

Búsqueda Distribuida en una *Estructura DFS-Tree CSP*

En este capítulo se presenta una nueva técnica para la resolución de CSPs distribuidos (DCSP). En los siguientes apartados se describe la idea principal del algoritmo distribuido, basado en una jerarquía DFS-Tree que aprovecha una estructura de árbol para la resolución en paralelo de los sub-problemas que recaen en diferentes ramas del árbol. Además se presentan dos métodos de resolución *intra-agente* adaptados a cada una de las técnicas de partición presentadas en el Capítulo 4, mediante los cuales cada "*agente*"puede realizar la búsqueda de *soluciones_parciales* a su sub-problema.

5.1. Notación

En esta sección se presentan algunas definiciones necesarias para la comprensión del método propuesto para la resolución de DCSP.

Definición 5.1.1. Una ***Estructura DFS-Tree CSP*** es un árbol cuyos nodos se componen de sub-problemas, donde cada sub-problema es de nuevo un CSP (sub-CSPs) y los arcos indican las relaciones entre los sub-problemas. La Figura 5.2-derecha representa una *Estructura DFS-Tree CSP*.

Definición 5.1.2. Se define como ***nodo-DFS*** a cada uno de los nodos de una *Estructura DFS-Tree CSP*. Como hemos indicado antes, un *nodo-DFS* es un sub-CSP, por lo que llamaremos ***nodo-individual*** a cada nodo atómico que forma parte de un sub-CSP; cada *nodo-individual* representa a una variable.

Definición 5.1.3. Una ***solución_parcial*** es una solución a un sub-CSP la cual satisface todas las *intra-restricciones* del sub-CSP y las *inter-restricciones* (si existen) entre este sub-CSP y sus sub-CSPs antecesores según la *Estructura DFS-Tree CSP*.

Definición 5.1.4. Una ***vista_parcial*** es un conjunto de pares variable-valor que contiene en cada *agente* el conjunto de asignaciones de variables que actualmente tienen sus *agentes* antecesores según la *Estructura DFS-Tree CSP*.

5.2. Creación de la jerarquia DFS-Tree

Cualquier CSP binario puede ser transformado en una *Estructura DFS-tree CSP*. Si el CSP original representa un problema inherentemente distribuido, cada uno de los sub-problemas identificados será englobado en un único *nodo-DFS*. Sin embargo, si tratamos con problemas particionables, será necesario realizar primero la partición del problema usando, por ejemplo, alguna de las técnicas presentadas en el capítulo anterior; cada uno de los sub-problemas resultantes de la partición será asociado a un único *nodo-DFS*. De esta manera, cada *nodo-DFS* contiene un sub-problema y debe ser resuelto por un *agente*.

El Algoritmo 2 nos muestra el proceso que seguimos para construir una *Estructura DFS-Tree CSP*. Los nodos y aristas del grafo G son respectivamente los *nodo-DFSs* y las *inter-restricciones* resultantes de la identificación/partición de sub-problemas. El *agente* raíz puede ser seleccionado aleatoriamente o bien aplicando algún criterio de restringibilidad (mayor número de variables, mayor número de restricciones, restricciones más duras,...). El algoritmo *DFSEstructura(G,v)* simplemente introduce el *nodo-DFS* v en la *Estructura DFS-Tree CSP*, marca el nodo para que pueda ser identificado como visitado, selecciona un nuevo *nodo-DFS* i entre todos sus *nodo-DFSs* adyacentes y llama recursivamente a *DFSEstructura(G,i)*. Si un *nodo-DFS* tiene varios *nodo-DFS* adyacentes, será igualmente correcto seleccionarlos en cualquier orden. Sin embargo, es muy importante retrasar la comprobación de si el *nodo-DFS* está o no visitado hasta que terminen las llamadas recursivas de los *nodo-DFSs* previamente seleccionados. La *Estructura DFS-Tree CSP*

resultante es usada como una jerarquía de comunicación entre los *agentes*, guiándolos en el proceso de envío y recepción de mensajes.

Algoritmo DFSEstructura(*G,v*) *;Originalmente ningún nodo está visitado,*
;la Estructura_DFS-Tree_CSP está vacía
;y v es el primer nodo-DFS de G.

begin

insertar(v,Estructura_DFS-Tree_CSP);

marcar v como visitado;

for each *nodo-DFS i adyacente* a v no visitado **do**

DFSEstructura(G,i); ;*nodo-DFS i* es adyacente a *nodo-DFS v* si al
;menos existe una *inter-restricción* entre i y v.

return *Estructura_DFS-Tree_CSP*.

end DFSEstructura

Algoritmo 2. Algoritmo DFSEstructura.

Ejemplo de *Estructura DFS-Tree CSP*

La Figura 5.1 muestra dos representaciones diferentes de un mismo CSP generado con el módulo $generate_R(C, n, k, p, q)$[1], un generador de CSPs aleatorios; donde C es la red de restricciones; n es el número de variables; k es el número de valores en cada dominio; p es la probabilidad de una arista no-trivial; q es la probabilidad de un par de valores válido en cada restricción. Esta figura representa la red de restricciones $< C, 20, 50, 0.1, 0.1 >$. Este problema tiene una gran dificultad para ser resuelto con algunos conocidos métodos de resolución de CSPs: Forward-Checking (FC) y Backjumping (BJ). El algoritmo FC, por ejemplo, no es capaz de encontrar una solución tras varias horas de ejecución.

[1]Una librería de rutinas para experimentar con diferentes métodos de backtracking para la resolución de CSPs http://ai.uwaterloo.ca/ vanbeek/software/software.html

Se puede observar cómo este problema puede ser dividido en varios subproblemas (ver Figura 5.2) y se puede convertir en una *Estructura DFS-Tree CSP* (ver Figura 5.2).

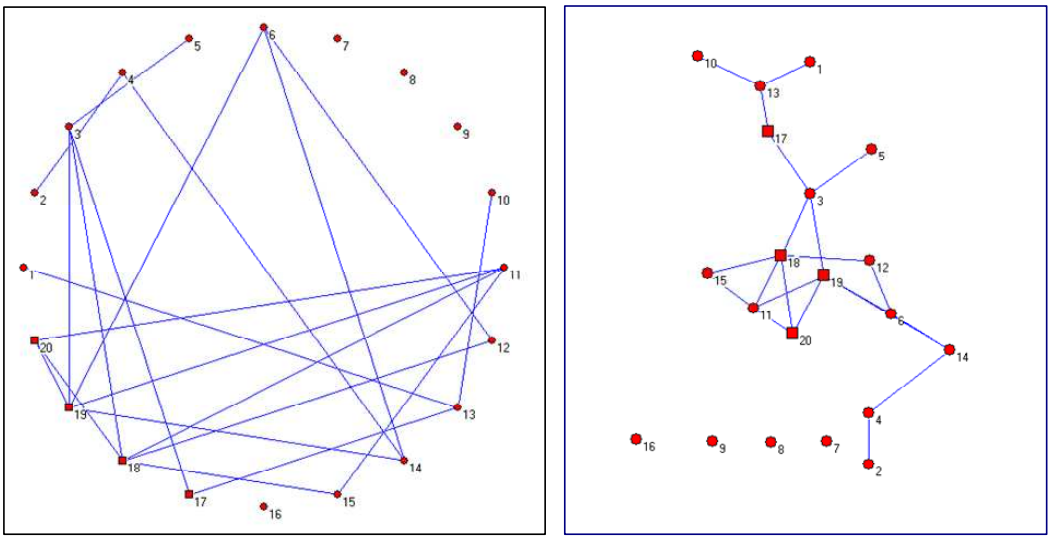


Figura 5.1: CSP aleatorio $< n = 20, d = 50, p = 0.1, q = 0.1 >$.

En la Figura 5.3 se puede observar una secuencia específica de nodos (orden numérico). Siguiendo esta secuencia, el algoritmo FC tiene un gran inconveniente debido a las variables no conectadas con alguna variable previa: 1, 2, 3, 6, 7, 8, 9, 10, 11 y 16 (ver Figura 5.3). Estas variables no tienen su dominio acotado por ninguna variable previa, lo que provoca la exploración de todo su dominio cuando el algoritmo vuelve atrás. Este tipo de variables es un caso extremo de variables con su dominio débilmente acotado. Un ejemplo de esta situación se puede ver en la Figura 5.3: la variable 12 tiene su dominio acotado por la variable 6. Cuando la variable 12 no tiene una asignación válida, el algoritmo FC vuelve atrás para cambiar la asignación de la variable 6, sin embargo, este algoritmo (FC) necesitará examinar completamente los dominios de las variables 11, 10, 9, 8 y 7 antes de conseguir cambiar la asignación de la variables 6. En este ejemplo, con una talla de $dominio = 50$, esta situación implica 50^5 asignaciones en vano.

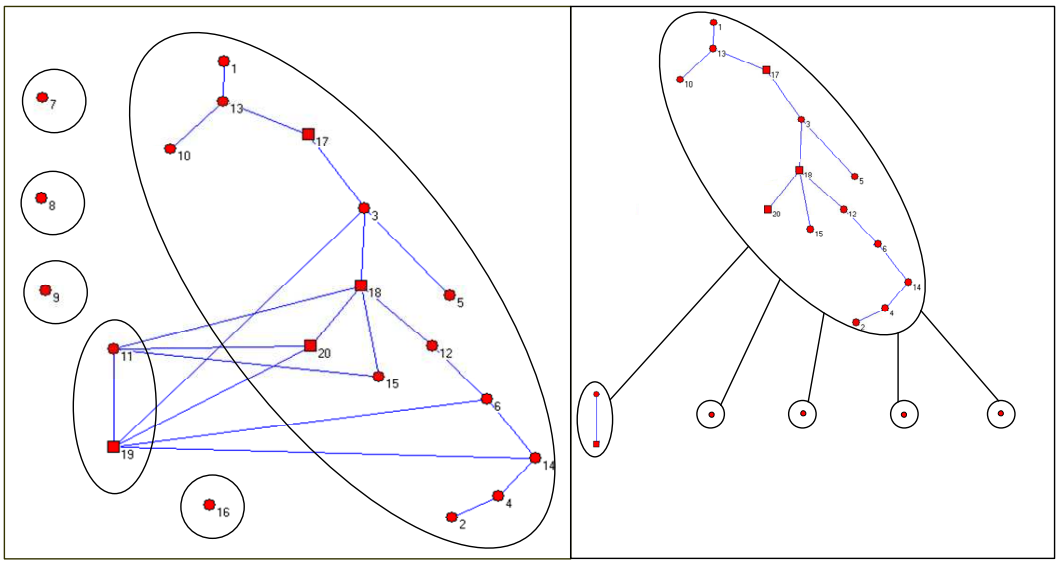


Figura 5.2: Izquierda: Problema particionado. Derecha: *Estructura DFS-Tree CSP*.

5.3. Algoritmo de resolución para DCSP

En este apartado presentamos el algoritmo DFSTreeSearch (DTS), Búsqueda Distribuida Basada en *Depth-First Search Tree* (*DFS-Tree*), al cual nos referiremos con el acrónimo DTS. Este algoritmo puede considerarse como una técnica asíncrona distribuida. *DFS-Tree* ya ha sido investigado como un método para potenciar la búsqueda [31]. Debido a la relativa independencia de los nodos situados en ramas diferentes del *DFS-Tree*, es posible realizar la búsqueda en paralelo en las ramas independientes.

DTS resuelve un CSP con técnicas distribuidas mediante una *Estructura DFS-Tree CSP*, donde un grupo de *agentes* manejan cada sub-CSP con sus variables (*nodo-individuales*) y sus *intra-restricciones* (aristas). Cada *agente* debe resolver su propio sub-problema compuesto por su sub-CSP y restringido por la asignación de valores generada por los *agentes* antecesores en la *Estructura DFS-Tree CSP*.

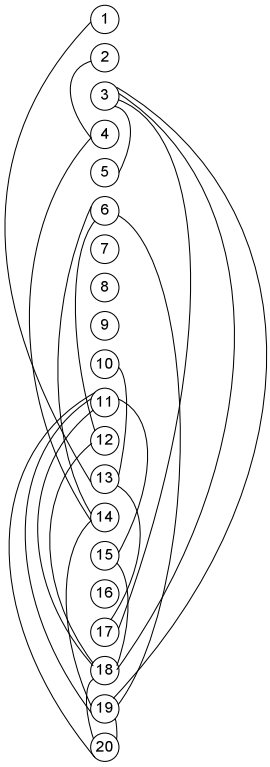


Figura 5.3: Secuencia de nodos seguida por el algoritmo Forward-Checking.

5.3.1. Algoritmo DFSTreeSearch (DTS)

El algoritmo comienza cuando el *agente* raíz intenta resolver su propio sub-problema (*nodo-DFS* raíz). Si el *agente* raíz encuentra una solución a su sub-problema, entonces envía su *vista_parcial* (la asignación de sus variables) a sus *agentes* hijos, según la *Estructura DFS-Tree CSP*, mediante un mensaje *START*. Ahora, todos sus hijos trabajan concurrentemente para resolver sus propios sub-problemas teniendo en cuenta la asignación parcial realizada por su padre (*agente* raíz). Cuando un *agente* hijo encuentra una solución a su sub-problema añade su propia solución a su *vista_parcial* (la asignación de sus variables y las de sus antecesores) y envía su *vista_parcial* a sus *agentes* hijos, el proceso continúa igual mientras no se alcance un *agente* hoja (*agente* sin hijos). Cuando un *agente* hoja encuentra solución a su sub-problema, éste envía un mensaje *OK* a su *agente* padre. Cuando un *agente*, tras enviar su

última *vista_parcial*, recibe un mensaje *OK* de todos sus hijos entonces envía un mensaje *OK* a su *agente* padre. Cuando el *agente* raíz recibe un mensaje *OK* de todos sus hijos, el algoritmo finaliza ya que se ha encontrado una solución al problema global.

Cuando un *agente* hijo no encuentra solución a su sub-CSP, envía un mensaje *Nogood* a su *agente* padre. El mensaje *Nogood* contiene un conjunto de variables cuya asignación imposibilita que el *agente* hijo encuentre una solución a su sub-CSP. Cuando un *agente* padre recibe un mensaje *Nogood*, primero detiene la búsqueda de todos sus hijos enviándoles un mensaje *STOP*, luego comprueba si alguna de las variables que contiene el mensaje *Nogood* le pertenecen; si ninguna variable del mensaje *Nogood* le pertenece, reenvía este mismo mensaje a su *agente* padre; de igual forma, si alguna variable del mensaje *Nogood* le pertenece comienza de nuevo el proceso intentando buscar una nueva solución a su sub-CSP; si encuentra dicha solución, cambia su *vista_parcial* y se la envía a sus hijos con un mensaje *START*; si no encuentra solución, envía un mensaje *Nogood* a su padre, este mensaje *Nogood* contiene un conjunto de variables cuya asignación imposibilita encontrar una solución a éste *agente* o a alguno de sus hijos. Si el *agente* raíz recibe un mensaje *Nogood* y tras procesarlo es incapaz de encontrar una solución a su sub-CSP, entonces el algoritmo termina porque el problema global no tiene solución.

Los Algoritmos 3, 4, 5 y 6 muestran los procesos para la ejecución del algoritmo DTS. Durante la ejecución del algoritmo DTS son cuatro los tipos de mensajes intercambiables: *OK*, *Nogood*, *START* y *STOP*; las acciones que cada *agente* lleva a cabo cuando recibe uno de estos mensajes se describen en los Algoritmos 3 y 4. El algoritmo comienza cuando el *agente* raíz recibe un mensaje *START* con el campo *vista_parcial* vacío por no tener ningún antecesor. Los Algoritmos 5 y 6 describen las principales funciones auxiliares usadas por el algoritmo DTS.

```
;Algoritmo DFSTreeSearch(Estructura_DFS-Tree_CSP)

when recibe(START,vista_parcial) do
    sol_parcial=buscarSolucion(agente_id,vista_parcial,Ø);
    if sol_parcial ≠ Ø
        insertar(sol_parcial,vista_parcial_agente_id);
        if hijos_id==Ø      ;el agente no tiene hijos
            if padre_id==Ø      ;agente raíz
                return SOLUCION;      ;FIN DTS
            else
                enviarMensaje(padre_id,OK,agente_id,vista_parcial_agente_id);
        else
            for each hijo i
                estado[i]=pendiente;
                enviarMensaje(i,START,vista_parcial_agente_id);
    else
        backtrack(agente_id);

when recibe(OK,hijo_id, vista_parcial) do
    if vista_parcial coincide con vista_parcial_agente_id
        estado[hijo_id]=satisfecho;
        when todos mis hijos i tienen estado[i]==satisfecho do
            if padre_id==Ø      ;agente raíz
                return SOLUCION;      ;FIN DTS
            else
                enviarMensaje(padre_id,OK,agente_id,vista_parcial_agente_id);
```

Algoritmo 3. Algoritmo DFSTreeSearch. Parte I.

Cuando un *agente* recibe un mensaje *START* (Algoritmo 3), primero llama a la función *buscarSolucion(agente_id,vista_parcial,Ø)* (Algoritmo 5) para buscar una *solución_parcial* a su sub-CSP. En esta llamada, la función *buscarSolucion* tiene el campo *mensaje_nogood* vacío ya que todavía el *agente* no ha recibido ningún mensaje *Nogood*. Entonces, la función *buscarSolucion* actualiza la *vista_parcial* del *agente* con los valores de la *vista_parcial* recibida en el mensaje *START*, esto lo realiza la función *actualizarVistaParcial*. A continuación se acota el dominio de las variables del *agente* (V_{agente_id}) de acuerdo a los valores de su *vista_parcial* y las *inter_restricciones* que dicho *agente* tiene con sus *agentes* antecesores. Esto se hace llamando a la función *restringirDominioVariables*. Esta función puede devolver un conjunto de variables (V_{nogood}), pertenecientes a alguno de los *agentes* antecesores, cuyas asignaciones dejan vacío el dominio de alguna de las variables del *agente*, por lo que el *agente* no podrá encontrar una *solución_parcial* acorde a su actual *vista_parcial*; en este caso la función *buscarSolucion* no devuelve ninguna solución. Sin embargo, si la función *restringirDominioVariables* no deja ningún dominio vacío, la función *buscarSolucion* intentará buscar una *solución_parcial* a su sub-CSP invocando a la función *busquedaSolucion*, la cual puede usar cualquier método para resolver el sub-CSP y devolver una *solución_parcial* si la encuentra o un conjunto vacío si no encuentra solución; el valor devuelto por esta función será a su vez retornado por la función *buscarSolucion* terminando así su ejecución.

Si durante el procesamiento de un mensaje *START* la función *buscarSolucion* devuelve una *solución_parcial*, el *agente* actualizará su *vista_parcial* con los nuevos valores de sus variables. Si se trata de un *agente* hoja (no tiene *agentes* hijos), y no es el *agente* raíz, el *agente* enviará un mensaje *OK* a su *agente* padre, que contiene aquellas variables de su *vista_parcial* que no pertenecen a su sub-CSP; sin embargo, si se trata del *agente* raíz y no tiene *agentes* hijos, el algoritmo terminará porque ya se ha encontrado una solución al problema. Por el contrario, si el *agente* sí que tiene *agentes* hijos, marcará el estado de todos ellos como *pendiente*, para indicar que todavía no ha recibido una respuesta de ellos, y les mandará a cada uno de ellos un mensaje *START*

con aquellas variables de su *vista_parcial* incluidas en alguna *inter-restricción* relacionada con algún *agente* descendiente del actual. Si por lo contrario, la función *buscarSolucion* no devuelve ninguna solución el *agente* ejecutará el procedimiento *backtrack*.

Cuando un *agente* recibe un mensaje *OK* (Algoritmo 3) de alguno de sus *agentes* hijos, comprueba que la *vista_parcial* del hijo_id coincide con la suya, es decir, las variables que tienen en común ambas *vistas_parciales* tienen los mismos valores asignados. Si coinciden, marcará el estado de este hijo_id como *satisfecho*, para indicar que su actual *vista_parcial* es consistente con el sub-CSP de su hijo_id. A continuación comprueba si todos sus hijos ya le han respondido positivamente; si es así, y se trata del *agente* raíz, el algoritmo DTS termina porque ya ha encontrado una solución al problema global; si no se trata del *agente* raíz, entonces el *agente* envía un mensaje *OK* a su *agente* padre, para indicarle que su sub-CSP y el de todos sus *agentes* hijos es consistente con la última *vista_parcial* que recibió, para ello el mensaje *OK* contiene aquellas variables de su *vista_parcial* que no pertenecen a su sub-CSP.

Cuando un *agente* recibe un mensaje *Nogood* (Algoritmo 4) de alguno de sus *agentes* hijos, comprueba que el *mensaje_nogood* coincide con su *vista_parcial*, es decir, las variables que tienen en común el *mensaje_nogood* y su *vista_parcial* tienen los mismos valores asignados. Si coinciden, marca el estado del hijo_id como *insatisfecho*, para indicar que el sub-CSP del *agente* hijo_id no es consistente con su actual *vista_parcial*. A continuación manda un mensaje *STOP* a todos sus hijos marcados con estado *pendiente*, es decir, a todos aquellos hijos de los que todavía no ha recibido ningún tipo de mensaje; el mensaje *STOP* enviado contiene su actual *vista_parcial*. Si el *mensaje_nogood* contiene alguna de las variables del sub-CSP del actual *agente*, entonces el *agente* llama a la función *buscarSolucion(agente_id,vista_parcial, mensaje_nogood)* para buscar una nueva *solución_parcial*. En esta ocasión el campo *mensaje_nogood* no está vacío, por lo que la función *buscarSolucion* primero llama a la función *podarVariablesNogood* (Algoritmo 6) para acotar, si es posible, el dominio de alguna variable. La función *podarVariablesNogood*

elimina, definitivamente, del dominio de una variable su valor actual cuando ésta es la única variable del *mensaje_nogood*. Sin embargo, cuando en el mensaje solamente hay una variable del actual sub-CSP pero hay más variables de los sub-CSPs de otros *agentes*, la función *podarVariablesNogood* elimina, del dominio de la variable contenida en el *mensaje_nogood* y que pertenece al actual sub-CSP, su valor actual, pero solamente mientras el *agente* padre del actual *agente* no cambie su *vista_parcial*. A continuación la función *buscarSolucion* intentará buscar una nueva *solución_parcial* a su sub-CSP invocando a la función *busquedaSolucion*, la cual puede usar cualquier método para resolver el sub-CSP y devolver una *solución_parcial* si la encuentra o un conjunto vacío si no encuentra solución. El valor devuelto por esta función será a su vez retornado por la función *buscarSolucion* terminando así su ejecución. Si la función *buscarSolucion* devuelve una *solución_parcial*, el *agente* actualizará su *vista_parcial* con los nuevos valores de sus variables. A continuación, el *agente* marcará el estado de todos sus hijos como *pendiente*, para indicar que todavía no ha recibido una respuesta de ellos, y les mandará a cada uno de ellos un mensaje *START* con aquellas variables de su *vista_parcial* incluidas en alguna *inter-restricción* relacionada con algún *agente* descendiente del actual. Si por lo contrario, la función *buscarSolucion* no devuelve ninguna solución el *agente* ejecutará el procedimiento *backtrack*.

Si el procedimiento *backtrack* (Algoritmo 5) es invocado por el *agente* raíz, el algoritmo DTS termina sin encontrar solución. Si por el contrario, es cualquier otro *agente* el que invoca el procedimiento *backtrack*, entonces se genera un *mensaje_nogood* que contiene un subconjunto de variables, de la *vista_parcial* del *agente*, que es inconsistente con las variables del actual sub-CSP, es decir, los valores asignados a este subconjunto de variables impiden que el *agente* actual encuentre una solución a su sub-CSP. Este *mensaje_nogood* es enviado al *agente* padre del actual *agente*.

```
;Algoritmo DFSTreeSearch(Estructura_DFS-Tree_CSP)

when recibe(Nogood,hijo_id,mensaje_nogood) do
    if mensaje_nogood coincide con vista_parcial_agente_id
        estado[hijo_id]=insatisfecho;
        for each hijo i con estado[i]==pendiente
            enviarMensaje(i,STOP,vista_parcial_agente_id);
        if V_agente_id ⋂ V_nogood ≠ ∅
            sol_parcial=buscarSolucion(agente_id,vista_parcial,mensaje_nogood);
            if sol_parcial ≠ ∅
                insertar(sol_parcial,vista_parcial_agente_id);
                for each hijo i
                    estado[i]=pendiente;
                    enviarMensaje(i,START,vista_parcial_agente_id);
            else
                backtrack(agente_id);
        else
            enviarMensaje(padre_id,Nogood,agente_id,mensaje_nogood);

when recibe(STOP,vista_parcial) do
    if vista_parcial coincide con vista_parcial_agente_id
        for each hijo i con estado[i]==pendiente
            enviarMensaje(i,STOP,vista_parcial_agente_id);
        reset(agente_id);      ;detiene cualquiera de sus procesos en ejecución
```

Algoritmo 4. Algoritmo DFSTreeSearch. Parte II.

```
;Algoritmo DFSTreeSearch(Estructura_DFS-Tree_CSP)
procedure backtrack(agente_id)
begin
    if padre_id==Ø      ;agente raíz
        return NO_SOLUCION;      ;FIN DTS
    else
        mensaje_nogood = {V_nogood|V_nogood = un subconjunto inconsistente
                                              de la vista_parcial del agente_id};
        enviarMensaje(padre_id,Nogood,agente_id,mensaje_nogood);
end backtrack

procedure buscarSolucion(agente,vista_parcial, mensaje_nogood)
begin
    if mensaje_nogood == Ø
        actualizarVistaParcial(agente,vista_parcial);
        V_nogood=restringirDominioVariables(agente);
        ;acota el dominio de las variables del agente
        ;según sus inter-restricciones y su vista_parcial
        if V_nogood ≠ Ø                    ;resuelve el sub-CSP
            solucion=busquedaSolucion();   ;con el algoritmo deseado
        else
            solucion= Ø;
    else
        podarVariablesNogood(V_agente, mensaje_nogood);
        solucion=busquedaSolucion();
    return solucion;
end buscarSolucion
```

Algoritmo 5. Algoritmo DFSTreeSearch. Parte III.

```
;Algoritmo DFSTreeSearch(Estructura_DFS-Tree_CSP)
procedure podarVariablesNogood(ConjuntoVar, mensaje_nogood)
begin
    for each X_i / (X_i ∈ mensaje_nogood) do
        if no existe X_j / (X_j ∈ mensaje_nogood)
            eliminarValorVariable(X_i);
        else if (X_i ∈ ConjuntoVar) ∧
                (no existe X_j / ((X_j ∈ mensaje_nogood) ∧ (X_j ∈ ConjuntoVar)))
            acotarValorVariable(X_i);
end podarVariablesNogood

end DFSTreeSearch
```

Algoritmo 6. Algoritmo DFSTreeSearch. Parte IV.

Cuando un *agente* recibe un *mensaje_Nogood* (Algoritmo 4) que no contiene ninguna de las variables de su sub-CSP, entonces el *agente* reenvía el mismo *mensaje_nogood* recibido a su *agente* padre.

Cuando un *agente* recibe un mensaje *STOP* (Algoritmo 4) que contiene una *vista_parcial* que coincide con la suya, el *agente* envía un nuevo mensaje *STOP* con su actual *vista_parcial* a todos sus hijos con estado *pendiente* y detiene su propia ejecución a la espera de recibir un nuevo mensaje de su padre.

La Figura 5.4 muestra un sencillo ejemplo donde tras estructurar un CSP en una *Estructura DFS-Tree CSP* se aplica el algoritmo DTS para buscar una solución al problema. El *agente* raíz (a_1) comienza el proceso de búsqueda encontrando una solución a su sub-problema. Entonces, envía esta *solución_parcial* a sus hijos. En este momento (estado 4), todos los hijos del *agente* raíz (a_2 y a_4) deben trabajar concurrentemente para encontrar una solución a sus propios sub-problemas y seguir así el proceso de búsqueda enviando de nuevo

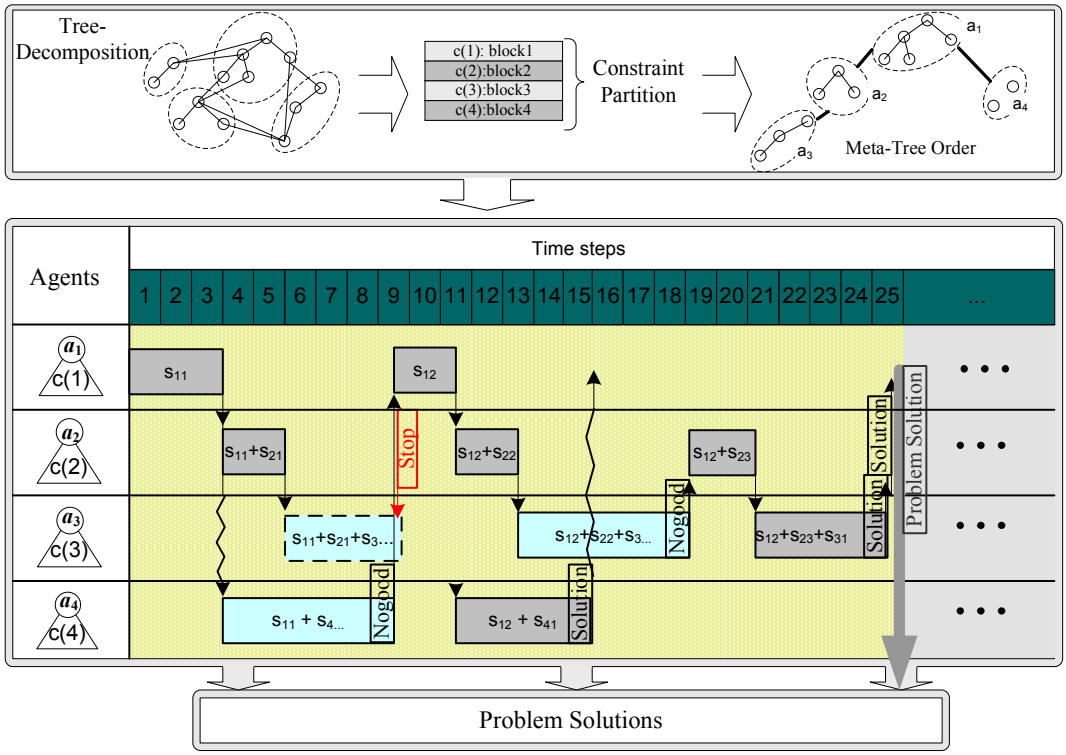


Figura 5.4: Técnicas de partición y búsqueda DTS.

sus *soluciones_parciales* (si las encuentran) a sus hijos (si los tienen). La solución al problema se encuentra cuando todos los *agentes* hoja alcanzan una solución a sus sub-problemas, por ejemplo, una solución al problema sería la asignación $(s_{12} + s_{41})$ del *agente* hoja a_4 más la asignación $(s_{12} + s_{23} + s_{31})$ del *agente* hoja a_3 (estado 25). En el estado 9 el *agente* a_4 envía un mensaje *Nogood* a su padre porque no es capaz de encontrar una asignación válida a su sub-problema. Entonces el *agente* padre a_1 detiene el proceso de búsqueda de todos sus descendientes y luego busca una nueva solución a su sub-problema, la cual enviará de nuevo a todos sus hijos (estado 12). Ahora, con la nueva asignación del *agente* a_1, el *agente* a_4 es capaz de encontrar una solución a su sub-problema y el *agente* a_2 trabaja con su hijo, el *agente* a_3, para encontrar las soluciones a sus sub-problemas. Cuando el *agente* a_3, en el estado 25,

encuentra una solución a su sub-problema, todos los *agentes* hoja han encontrado una solución a sus sub-problemas, por lo que ya tenemos la solución al problema global.

5.3.2. Ejemplo de ejecución del Algoritmo DTS

Veamos en la Figura 5.5 un ejemplo para analizar el comportamiento del algoritmo DTS. Primero la red de restricciones de la Figura 5.5(1) se particiona en tres árboles y se construye una *Estructura DFS-Tree CSP* (Figura 5.5(2)). El *agente a* encuentra su primera *solución_parcial* ($X_1 = 1, X_2 = 1$) y la envía a sus hijos: los *agentes b* y *c* (Figura 5.5(3)). Esta asignación parcial es una buena solución para el *agente c* que logra encontrar una solución a su sub-problema (Figura 5.5(4)), sin embargo el *agente b* no logra encontrar ninguna asignación compatible con la *solución_parcial* de su padre por lo que envía un mensaje *Nogood* al *agente a* (*Nogood* ($X_1 = 1$)) (Figura 5.5(5)). Entonces, el *agente a* procesa el mensaje *Nogood*, poda su espacio de búsqueda, busca una nueva solución ($X_1 = 2, X_2 = 2$) y la envía a sus hijos (Figura 5.5(6)). Ahora es el *agente c* quien envía un mensaje *Nogood* a su padre (*Nogood* ($X_1 = 2$)) porque la asignación que el padre realizó deja el dominio de la variable X_5 vacío (Figura 5.5(7)). El *agente a* detiene la búsqueda del *agente b* (Figura 5.5(8)) y luego procesa el mensaje *Nogood*, poda su espacio de búsqueda, encuentra una nueva solución a su sub-problema ($X_1 = 3, X_2 = 3$) y la envía a sus hijos (Figura 5.5(9)). Esta última *solución_parcial* es buena para ambos *agentes* hijos, por lo que ambos responden con un mensaje *OK* a su padre y la búsqueda termina (Figura 5.5(10)).

5.3.3. Algoritmo DTS: correcto y completo

Si hay una solución, el algoritmo alcanza un estado estable donde todas las variables satisfacen todas las restricciones y el algoritmo termina; esta situación se produce cuando el *agente* raíz recibe un mensaje *OK* de todos sus *agentes* hijos. Si no existe una solución, el algoritmo detecta esta situación y termina. Esto se produce cuando el *agente* raíz recibe un mensaje *Nogood* y no es capaz

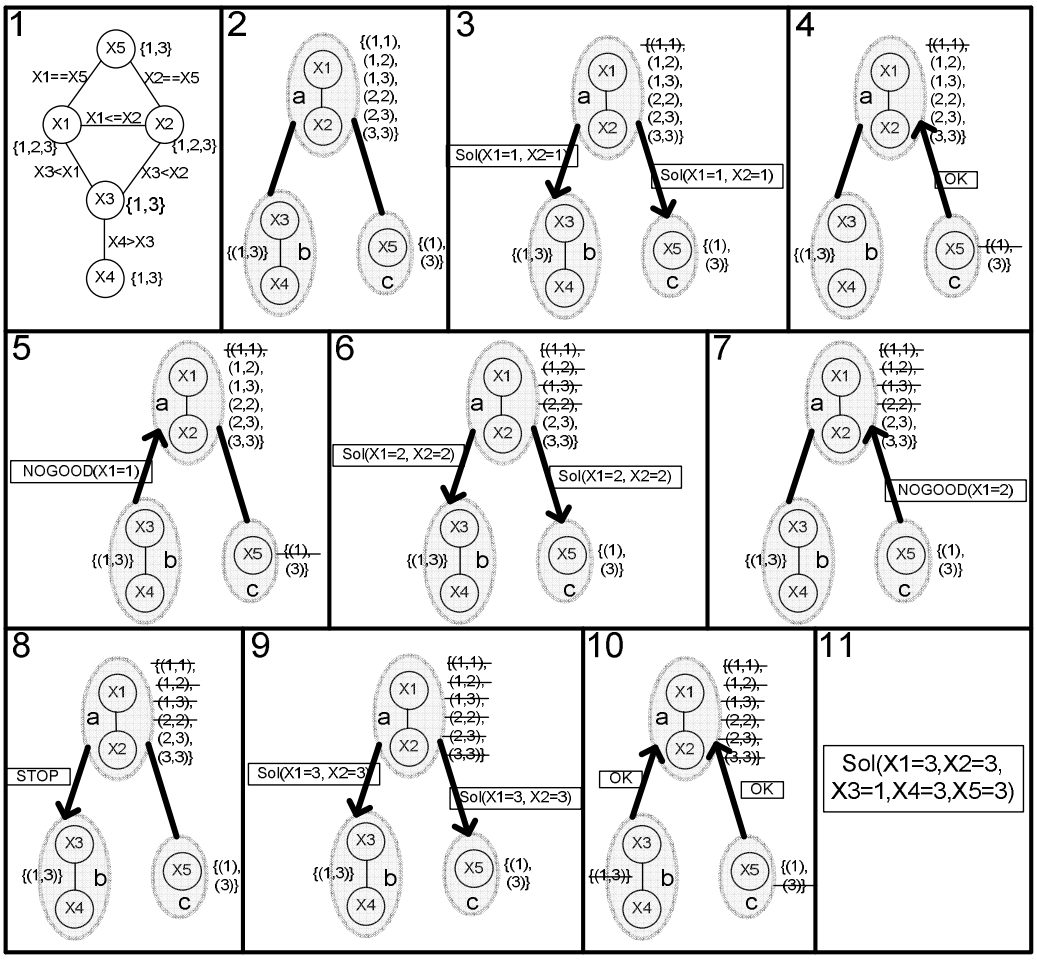


Figura 5.5: Ejemplo de ejecución del algoritmo DTS.

de encontrar una nueva solución a su sub-problema. Por ejemplo, en la Figura 5.5(10), si en el estado 10 uno de los *agentes* b o c enviasen un mensaje *Nogood*, entonces el *agente* raíz a tendría su dominio vacío y terminaría el proceso con "*NO_SOLUCION*". Por lo tanto el algoritmo es correcto y completo ya que encuentra una solución correcta cuando existe y termina sin solución cuando no existe ninguna solución.

Lo que falta por demostrar es que DTS alcanza uno de estos estados en tiempo finito. La única manera en la que DTS podría no terminar es si al menos un *agente* entrara en un bucle repitiendo cíclicamente la asignación

de los mismos valores a sus variables. A continuación demostraremos, por inducción, que esta situación nunca puede suceder.

En el caso base, asumimos que el *agente* raíz está en un ciclo infinito. Como él es el *agente* raíz, solamente recibe mensajes *Nogood* o *OK*. Cuando envía a sus hijos una posible *solución_parcial*, él puede recibir sólo dos tipos de respuestas:

1. *Respuesta_1*: todos los hijos responden con mensajes *OK*.

2. *Respuesta_2*: algún hijo responde con mensaje *Nogood*.

Si para todas las *soluciones_parciales* a su sub-problema el *agente* raíz recibe una respuesta de tipo *Respuesta_2* (es decir, algún hijo no puede encontrar una asignación válida a su sub-problema conforme a la *solución_parcial* del *agente* raíz), este *agente* terminará la búsqueda porque habrá agotado el dominio de todas sus variables y el algoritmo DTS terminará con "*solución no encontrada*". Si por el contrario, el *agente* raíz recibe una respuesta de tipo *Respuesta_1* (es decir, todos los *agentes* hijos encuentran una solución válida a su sub-problema acorde con la *solución_parcial* del *agente* raíz) entonces él no cambia su asignación parcial, porque es válida. Por lo tanto, en ninguno de los dos casos posibles el *agente* raíz puede entrar en un bucle infinito.

Ahora, asumimos que desde el *agente* raíz hasta el *agente* en el nivel $k-1$ del árbol ($k > 2$) mantienen un estado parcial consistente, y que el *agente* en el nivel k está en un bucle infinito. En este caso, si el *agente* tiene hijos, los únicos tipos de mensajes que el *agente* en el nivel k recibe de sus hijos son *Nogood* o *OK*. Por lo que las únicas respuestas que puede recibir de sus hijos son de nuevo *Respuesta_1* y *Respuesta_2*. Si el *agente* en el nivel k recibe siempre la respuesta tipo *Respuesta_2* el *agente* irá cambiando las instanciaciones de sus variables con diferentes valores para buscar nuevas *soluciones_parciales*, como el dominio de las variables es finito, el *agente* agotará todos los posibles valores de sus variables en un número finito de pasos y luego enviará un mensaje *Nogood* a su padre, por lo que contradice la suposición de que el *agente* en el nivel k estaba en un bucle infinito. Si por el contrario el *agente* del nivel

k recibiera una respuesta de tipo *Respuesta_1*, entonces confirmaría que su asignación es válida para sus hijos y respondería con un mensaje *OK* a su padre, de nuevo se contradice la suposición de que el *agente* en el nivel k estaba en un bucle infinito. Si el *agente* en el nivel k no tiene hijos, nunca recibirá mensajes *Nogood* o *OK*, por lo que nunca entrará en un bucle infinito, ya que se limitará a resolver su sub-problema y enviar una mensaje *OK* o *Nogood* a su padre, dependiendo de si encuentra o no una *solución_parcial* a su sub-problema. De este modo, por contradicción, el *agente* en el nivel k no puede estar en un bucle infinito. Así, por inducción, todos los *agentes* desde el nivel 1 hasta el nivel n, donde n es el nivel más bajo del árbol, terminan en tiempo finito.

5.4. Búsqueda *intra-agente* para CSPs Distribuidos

En el apartado anterior presentamos un algoritmo para la resolución de un CSP mediante técnicas distribuidas. De este modo cada *agente* tiene el compromiso de resolver su propio sub-problema hasta alcanzar una solución global al CSP completo. Siguiendo el algoritmo DTS (Algoritmos 3, 4, 5 y 6) cada *agente* es autónomo para resolver su propio sub-problema con el método de resolución que él considere más apropiado.

En este apartado vamos a presentar dos algoritmos para la resolución de un CSP centralizado que aprovechan la información recibida mediante el paso de mensajes *Nogood* realizado por el algoritmo distribuido DTS:

- El primero, al que llamaremos Forward Checking Acotado (FCA), está basado en el algoritmo FC y es un algoritmo aplicable a cualquier estructura de CSP;

- el segundo, llamado Tree Search Acotado (TSA), sin embargo es solamente aplicable a CSPs con estructura de árbol, ya que aprovecha la

característica de este tipo de estructuras para encontrar una solución al problema en tiempo polinómico.

5.4.1. Forward Checking Acotado (FCA)

En esta sección presentamos el algoritmo Forward Checking Acotado (FCA) [10]. FCA se basa en el algoritmo Forward Checking introducido en el Capítulo 2. Sin embargo, en este caso el algoritmo original ha sido modificado para hacer uso de la información proveniente de los mensajes *Nogood* recibidos de nuestros *agentes* hijos durante el proceso de búsqueda del Algoritmo Distribuido DTS. Los mensajes *Nogood* contienen conjuntos de asignaciones de variables que nos conducen a estados inconsistentes. Usando correctamente la información contenida en este tipo de mensajes es posible podar el espacio de búsqueda sin perder ninguna *solución_parcial* válida a nivel global, es decir, una *solución_parcial* que forma parte de una solución global del problema completo. Sin embargo, sí que eliminaremos *soluciones_parciales* consistentes con nuestro sub-problema, pero inconsistentes con el problema completo, es decir, *soluciones_parciales* que nunca formarán parte de una solución global del problema. Al igual que el algoritmo Forward Checking, FCA tiene una complejidad exponencial en el peor de los casos.

El Algoritmo 7 nos muestra el pseudocódigo del algoritmo FCA. Podemos observar que FCA se comporta exactamente igual que FC cuando todavía no hemos recibido ningún mensaje *Nogood*, esto es, cuando estamos buscando la primera solución a nuestro sub-problema. Sin embargo, el algoritmo se comporta de diferente manera cuando se trata de encontrar las siguientes soluciones a nuestro sub-problema.

Según el algoritmo distribuido DTS, un *agente* se ve en la tarea de buscar una nueva solución a su sub-problema cuando recibe un mensaje *Nogood* de alguno de sus hijos, conteniendo alguna de sus variables. La idea del algoritmo es podar el espacio de búsqueda, evitando la exploración de los dominios de todas aquellas variables con un orden superior (según la secuencia de búsqueda) al de la variable (X_j) con mayor orden incluida en el mensaje *Nogood*. De este

modo, no podaremos ninguna solución válida, ya que el espacio de búsqueda eliminado sería inconsistente con el mensaje *Nogood* recibido. Para ello el algoritmo FCA deshace los efectos de la asignación de todas las variables de orden superior o igual a X_j y continua aplicando el algoritmo FC básico partiendo de la variable X_j. Así nos aseguramos de que la nueva solución (si la hay) enviada a nuestros hijos cambiará el valor de al menos una de las variables contenidas en el último mensaje *Nogood*. Para poder asegurar esto tenemos que diferenciar dos casos:

- Si el mensaje *Nogood* contiene solamente una de las variables (X_j) del sub-CSP del actual *agente*, su valor habrá sido eliminado bien por la función *eliminarValorVariable(X_j)* o por la función *acotarValorVariable(X_j)* (en Algortimo 6). Ambas funciones eliminan el valor actual de la variable X_j de manera que ese valor no pueda volver a ser asignado a dicha variable; por lo tanto, al menos esta variable (X_j) cambiará su valor. En el primer caso (*eliminarValorVariable(X_j)*) el valor actual de la variable se elimina definitivamente del dominio de la variable X_j ya que ese valor siempre producirá una inconsistencia. Sin embargo, en el segundo caso (*acotarValorVariable(X_j)*) el valor actual de la variable sólo se elimina temporalmente ya que se sabe que es inconsistente con mi actual *vista_parcial*. Sin embargo, cuando mi *vista_parcial* cambie este valor podría volver a ser válido, por lo que de nuevo pertenecerá a D_j.

- Si el mensaje *Nogood* contenía 2 o más de las variables del sub-CSP del actual *agente*, el valor actual de X_j (variable con mayor orden incluida en el mensaje *Nogood*) no será eliminado del dominio de D_j, pero sí será marcado como acotado por el valor de X_s (segunda variable de mayor orden, según secuencia de búsqueda, que está incluida en el mensaje *Nogood*). De esta manera hasta que no cambie el valor de X_s, X_j no podrá volver a tomar su valor actual. Sin embargo, si al ejecutar el algoritmo FC(X_j) básico, el dominio de la variable X_s no tuviese más valores válidos, el algoritmo FC haría backtracking y podría devolvernos

una solución en la que no cambiase ningún valor de las variables contenidas en el último mensaje *Nogood*. Por esto, el algoritmo FCA no termina hasta que se asegura que algún valor de estas variables ha cambiado, ya que si no estaríamos enviando a nuestros hijos una *solución_parcial* distinta pero con la misma inconsistencia que provocó el anterior mensaje *Nogood*. Si no la encuentra, termina con *NO_SOLUCION*.

Algoritmo FCA(G, mensaje_nogood)

;G es el grafo de restricciones e incluye la secuencia de búsqueda d=(X_1,...,X_n)

begin

if *mensaje_nogood*==∅ *;buscamos la primera solución el sub-CSP*

solucion=FC(X_1); *;aplicar algoritmo Forward Checking a partir de* X_1

else

while (solucion ≠ NO_SOLUCION ∧ ninguna variable incluida en el mensaje *Nogood* cambie su valor)

if al menos 2 de mis variables pertenecen al mensaje *Nogood*

marcar valor(X_j) acotado por X_s;

;X_j es la variable de mayor orden, según d,

; que está incluida en el mensaje Nogood

;X_s es la segunda variable de mayor orden, según d,

; que está incluida en el mensaje Nogood

for each X_i / $(j \leq i < n)$

retractar los efectos de la asignación X_i←valor(X_i);

solucion=FC(X_j); *;aplicar algoritmo Forward Checking ;a partir de* X_j

return solucion;

end FCA

Algoritmo 7. Algoritmo Forward Checking Acotado.

Ejemplo de ejecución del algoritmo FCA

A continuación mostraremos un ejemplo de la ejecución del algoritmo FCA. El ejemplo está basado en el CSP Distribuido que muestra la Figura 5.6. En esta figura podemos ver un CSP particionado en 2 sub-problemas.

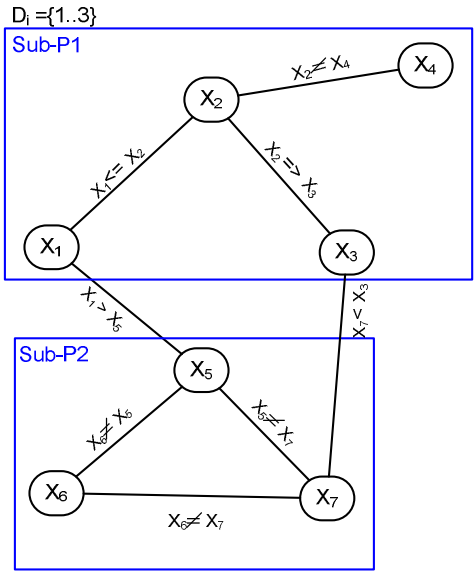


Figura 5.6: Ejemplo de CSP Distribuido.

El proceso para la resolución del CSP de la Figura 5.6 según el algoritmo DTS comenzaría con la búsqueda de la primera *solución_parcial* para el sub-problema 1. Para realizar esta búsqueda utilizaremos el algoritmo FCA. Como se trata de buscar la primera solución, el algoritmo FCA se comporta de igual manera que el algoritmo FC y encuentra la primera solución del conjunto de soluciones validas mostrado en la Figura 5.7: (X_1=1, X_2=1, X_3=1, X_4=2). Esta primera *solución_parcial* es inconsistente con el sub-problema 2, ya que la asignación X_1=1 deja vacío el dominio de X_5. Por lo tanto, se envía un mensaje *Nogood* ($X_1 = 1$).

Debido al algoritmo DTS, cuando el *agente* encargado de resolver el sub-problema 1 reciba el mensaje *Nogood* ($X_1 = 1$), eliminará de D_1 el valor 1, y buscará una nueva solución ejecutando el algoritmo FCA. FCA detecta la poda de todas las soluciones en las que X_1=1 (ver Figura 5.8), por lo que deshace

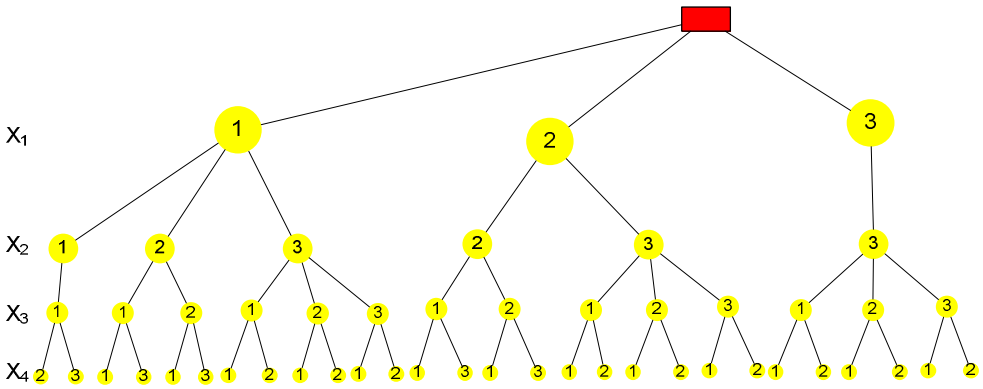


Figura 5.7: Conjunto de Soluciones válidas para Sub-P1 (Figura 5.6).

todos los efectos provocados por las asignaciones de las variables X_3, X_2 y X_1 y aplica el algoritmo FC(X_1) para encontrar una nueva *solución_parcial*: (X_1=2, X_2=2, X_3=1, X_4=1). De nuevo, la *solución_parcial* del sub-problema 1 es inconsistente con el sub-problema 2 porque $X_3 = 1$ deja el dominio D_7 vacío. Esta situación genera el mensaje *Nogood* (X_3=1).

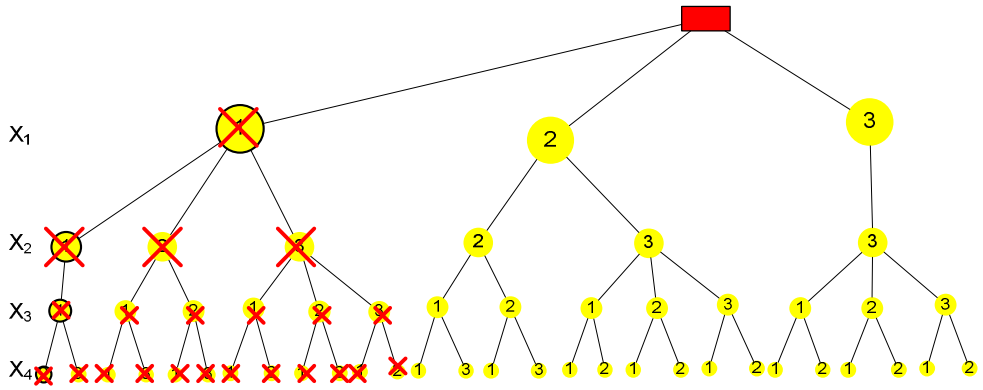


Figura 5.8: Poda del Algoritmo FCA tras recibir el *Nogood* (X_1=1).

El mensaje *Nogood* (X_3=1) provoca la poda de todas aquellas soluciones en las que X_3=1 (ver Figura 5.9). En esta ocasión cuando se ejecute FCA para buscar una nueva solución al sub-problema 1, FCA solamente retractará los efectos de la asignación de la variable X_3 y ejecutará FC(X_3) para buscar una nueva *solución_parcial*: (X_1=2, X_2=2, X_3=2, X_4=1). Tampoco esta nueva *solución_parcial* es consistente con el sub-problema 2, ya que la asignación de

las variables X_1=2 y $X_3 = 2$ deja a los dominios D_5 y D_7 con ningún valor consistente para satisfacer la restricción $X_5 \neq X_7$. Así se genera el mensaje *Nogood* (X_1=2,$X_3 = 2$).

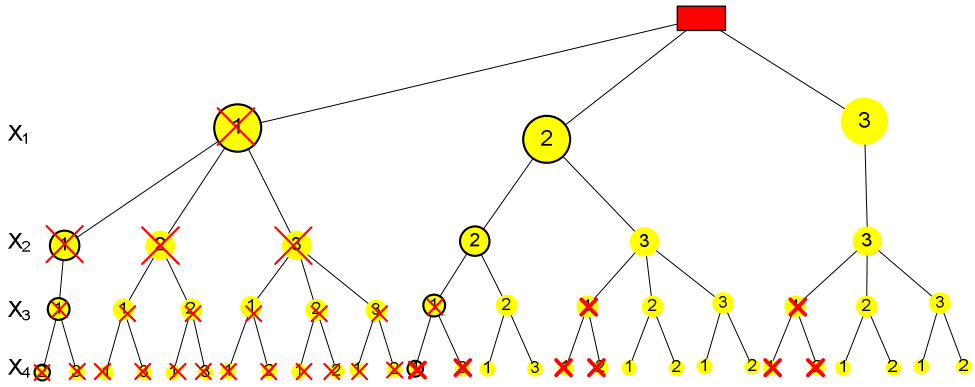


Figura 5.9: Poda del Algoritmo FCA tras recibir el *Nogood* (X_3=1).

La información contenida en el mensaje *Nogood* (X_1=2,$X_3 = 2$), provoca que el algoritmo FCA marque el dominio de X_3 acotado por la variable X_1. De este modo, hasta que no se cambie el valor de la variable X_1, la variable X_3 no podrá volver a tomar el valor 2. Esto provoca la poda de las soluciones mostradas en la Figura 5.10. FCA deshace los efectos de la asignación de la variable X_3 y ejecuta el algoritmo FC(X_3) que finalmente encontrará una *solución_parcial* consistente con el sub-problema 2: (X_1=2, X_2=3, X_3=3, X_4=1).

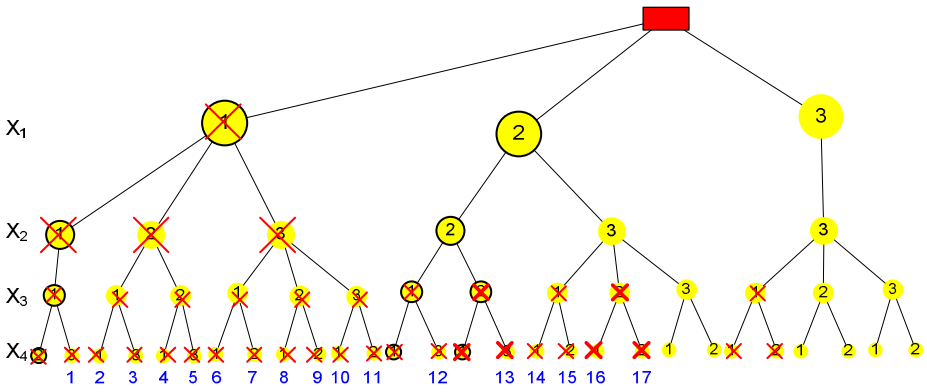


Figura 5.10: Poda del Algoritmo FCA tras recibir el *Nogood* (X_1=2,X_3=2).

En la Figura 5.10 podemos comprobar como gracias al procesamiento de la información recibida en los mensajes *Nogood* se han podado 17 *soluciones_parciales* del sub-problema 1 inconsistentes con el sub-problema 2, antes de encontrar la *solución_parcial* del sub-problema 1 que forma parte de la solución global del problema de la Figura 5.6.

5.4.2. Búsqueda acotada basada en estructuras de árbol

En esta sección presentamos el algoritmo Tree Search Acotado (TSA). TSA es un algoritmo para resolver CSPs con estructura de árbol. Este algoritmo se basa en el teorema presentado en [31] que dice: "Sea d una ordenación de ancho 1 de la red con estructura de árbol T. Si T tiene arco-consistencia direccional relativa a d, entonces la red es libre de backtracking a lo largo de d". Esto significa que seremos capaces de encontrar una solución sin necesidad de retroceder en el proceso de búsqueda si realizamos la asignación de las variables según el orden d. Por lo tanto, el algoritmo TSA nos permite encontrar una solución en tiempo polinomial.

Además TSA también hace uso de la información proveniente de los mensajes *Nogood* recibidos de nuestros *agentes* hijos durante el proceso de búsqueda del algoritmo distribuido DTS. Usando esta información es capaz de podar *soluciones_parciales* válidas para el sub-problema en cuestión, pero inconsistentes con las restricciones del problema global.

Los Algoritmos 8, 9, 10 y 11 nos muestra el pseudocódigo del algoritmo TSA; en los Algoritmos 8 y 9 vemos la función principal del algoritmo TSA y en los Algoritmos 10 y 11 las funciones auxiliares. El algoritmo recibe como datos de entrada una red con estructura de árbol y un orden de ancho 1 relativo a esa red. TSA actúa de distinto modo para encontrar la primera solución del sub-problema que para encontrar las sucesivas soluciones del mismo.

Para realizar la búsqueda de la primera solución (Algoritmo 8), lo primero que realiza el algoritmo TSA es una arco-consistencia direccional relativa a d, aplicando el algoritmo DAC presentado en la sección 2.3. Si la red tiene arco-consistencia direccional, ya solo resta realizar la asignación de las variables

(*asignarValor(*1*)*), siguiendo el orden d para asegurarnos que no tendremos que hacer ninguna reasignación de valores. Si la red no tiene arco-consistencia direccional, porque el algoritmo DAC ha dejado algún dominio vacío, entonces el sub-problema no tiene solución.

Algoritmo TSA(G, *mensaje_nogood*)

;G es el grafo de restricciones e incluye

;el orden de búsqueda d=(X_1*,...,*X_n*) que debe ser de ancho 1*

begin

if *primera_solucion* *;buscamos la primera solución*

DAC(G); *;aplicar Arco-Consistencia Direccional*

if $\forall X_i$: $D_i \neq \emptyset$

solucion=asignarValor(1);

else

return NO_SOLUCION;

1. **else** *;buscamos una nueva solución*

Algoritmo 8. Algoritmo Tree Search Acotado (TSA). Parte I.

Al igual que el algoritmo FCA, cuando ejecutamos TSA después de recibir un mensaje *Nogood* (Algoritmo 9), el algoritmo poda el espacio de búsqueda evitando la exploración de los dominios de todas aquellas variables con un orden superior (según la secuencia de búsqueda d) al de la variable (X_j) con mayor orden incluida en el mensaje *Nogood*. De este modo, no podaremos ninguna solución válida, ya que el espacio de búsqueda eliminado sería inconsistente con el problema global por no modificar los valores de ninguna de las variables incluidas en el último mensaje *Nogood*.

```
1.    else      ;buscamos una nueva solución
2.        marcarVariablesNogood(V_agente_id, mensaje_nogood);
3.        if al menos 2 de las variables del agente
              están incluidas en mensaje_nogood
4.            marcar valor(X_j) acotado por X_s;
              ;X_j es la variable de mayor orden, según d, incluida en mensaje_nogood
              ;X_s es la 2ª variable de mayor orden, según d, incluida en mensaje_nogood
5.            revisarArcoConsistenciaD(X_j,X_s);
6.        else
7.            revisarArcoConsistenciaD(X_j,X_1);
8.        solucion=SOL_NO_VALIDA;
9.        while solucion==SOL_NO_VALIDA do
10.           k=j;      ;antVarNogood contiene el orden de la 1ª variable anterior
11.           while k ≥ 1 ∧ D_k==∅ do ;a k marcada como varNogood
12.               k=antVarNogood;
              if k < 1      return NO_SOLUCION;
              else
                  for each X_i / (k ≤ i < n)
                      retractar los efectos de la asignación X_i=valor(X_i);
                  solucion=asignarValor(k);
                  if ninguna variable del mensaje_nogood cambia su valor
                      solucion=SOL_NO_VALIDA;
                      marcarVariablesNogood(V_agente_id, mensaje_nogood);
                      marcar valor(X_j) acotado por X_s;
                      revisarArcoConsistenciaD(X_j,X_s);
    return solucion;
end TSA
```

Algoritmo 9. Algoritmo Tree Search Acotado (TSA). Parte II.

```
;Algoritmo TSA(G, mensaje_Nogood)

procedure revisarArcoConsistenciaD(var-inicio, var-fin)
begin
    if (padre(var-inicio) < var-fin) ∧
       (revisar(padre(var-inicio),var-inicio,var-fin))
         revisarArcoConsistenciaD(padre(var-inicio),var-fin);
end revisarArcoConsistenciaD
procedure revisar(i,j,fin);Retorna True sii eliminamos valores de D_i
begin
    removed ← false;
    for each v in D_i do
        if ningún-valor y in D_j satisface c_ij
            marcar v acotado por X_fin;
            removed ← true;
    return removed;
end revisar
procedure asignarValor(i)
begin
    Instanciar X_i ← a_i : a_i ∈ D_i;
    for each X_j / X_j es hijo de X_i
        acotar de D_j aquellos valores inconsistente con respecto a la
        instanciación X_i ← a_i y la restricción c_ij.
    if i < n
        Desmarcar X_{i+1} como varNogood;
        asignarValor(i + 1);
end asignarValor
```

Algoritmo 10. Algoritmo Tree Search Acotado (TSA). Parte III.

```
;Algoritmo TSA(G, mensaje_Nogood)

procedure marcarVariablesNogood(ConjVar, mensaje_nogood)
begin
    for each X_i / (X_i ∈ ConjVar ∧ X_i ∈ mensaje_nogood ) do
        marcar X_i como varNogood;
        X_j=padre(X_i);
        while X_j ≠ Ø
            marcar X_j como varNogood;
            X_j=padre(X_j);
end marcarVariablesNogood
```

Algoritmo 11. Algoritmo Tree Search Acotado (TSA). Parte IV.

El primer paso que ejecuta el algoritmo TSA, después de recibir un mensaje *Nogood*, es marcar todas las variables involucradas en el mensaje *Nogood* (las variables del mensaje y sus correspondientes antecesores) como *varNogood*. A continuación TSA revisa la arco-consistencia direccional de su red de restricciones, ya que algún valor, de alguna de las variables incluidas en el mensaje *Nogood*, habrá sido eliminado o acotado. Para realizar esta comprobación tenemos que contemplar dos casos:

- Si el mensaje *Nogood* contenía solamente una de las variables (X_j) del agente, su valor habrá sido eliminado bien por la función *eliminarValorVariable(X_j)* o por la función *acotarValorVariable(X_j)* (su descripción en la sección 5.4.1), entonces la función *revisarArcoConsistenciaD*(X_j,X_1) (Algoritmo 10) debe revisar los dominios de las variables comprendidas entre X_1 (la variable raíz) y X_j (la variable contenida en el mensaje *Nogood* y cuyo valor ha sido eliminado) para asegurar que se siga cumpliendo la arco-consistencia direccional. No es necesario comprobar la arco-consistencia direccional de las variables

en un nivel del árbol inferior al de X_j, ya que la eliminación de un valor de una variable de nivel superior (X_j) no les afecta debido al orden seguido al aplicar la arco-consistencia direccional.

- Si el mensaje *Nogood* contenía 2 o más de las variables del agente, el valor actual de X_j (variable con mayor orden, según la secuencia de búsqueda d, incluida en el mensaje *Nogood*) no será eliminado del dominio de D_j, pero sí será marcado como acotado por el valor de X_s (segunda variable de mayor orden que está incluida en el mensaje *Nogood*). En esta situación la función *revisarArcoConsistenciaD*(X_j,X_s) (Algoritmo 10) solo deberá revisar la arco-consistencia direccional entre las variables comprendidas entre X_j y X_s, ya que cuando X_s cambie su valor actual, el valor de X_j acotado por X_s volverá a pertenecer al D_j.

La función *revisarArcoConsistenciaD* solamente revisa la arco-consitencia direccional en la rama del árbol afectada por la eliminación de un valor, ya que este hecho no afecta a ninguna de las otras ramas del árbol al no existir ninguna relación entre sus nodos.

El siguiente paso para buscar una nueva solución es asignar valores a las variables, para ello primero debemos determinar a partir de qué variable debemos reasignar valores de manera que se realice la máxima poda posible pero sin que se pierda ninguna *solución_parcial* consistente con la solución global. Como debemos cambiar alguno de los valores de las variables incluidas en el mensaje *Nogood* (para evitar una nueva *solución_parcial* con la misma inconsistencia que la anterior), podemos podar el espacio de búsqueda de todas aquellas variables con un orden, según d, mayor al de X_j. Por ello buscamos a partir de X_j, y en orden decreciente, la primera variable que tiene en su dominio valores válidos para asignar a su variable. En este proceso de búsqueda no se tienen en cuenta aquellas variables que no estén marcadas como *varNogood*, ya que o no están relacionadas con ninguna inconsistencia detectada hasta el momento, o sus valores ya han sido testeados o descartados después de detectarse la inconsistencia. Si no encontramos ninguna variable con valores válidos

disponibles, entonces TSA termina con NO_SOLUCION. Si alguna variable todavía tiene algún valor válido en su dominio (X_k), TSA deshace los efectos de las asignaciones de todas las variables con orden superior al de esta variable (X_k incluida) y después realiza la asignación de valores, en orden creciente según d, desde ella hasta X_n ($asignarValor(k)$).

Para asegurarse que la nueva solución no tendrá la misma inconsistencia que provocó el último mensaje *Nogood*, el algoritmo TSA no termina hasta que no encuentra una solución en la que cambien los valores de alguna de las variables incluidas en el último mensaje *Nogood*.

A continuación mostraremos un ejemplo de la ejecución del algoritmo TSA. El ejemplo está basado en el CSP Distribuido que muestra la Figura 5.11. En esta figura podemos ver un CSP particionado en 2 sub-problemas, cada uno de ellos representado con una estructura de árbol.

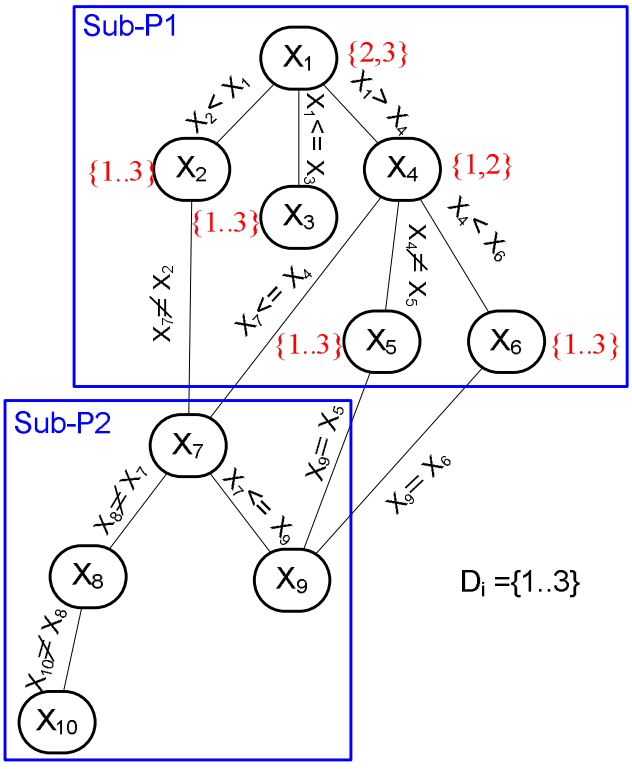


Figura 5.11: Ejemplo de CSP Distribuido con estructuras de árbol.

Para realizar la búsqueda de una solución para el problema global, los *agentes* encargados de resolver cada sub-problema seguirán el proceso de comunicación del algoritmo DTS y utilizarán el algoritmo TSA para buscar las *soluciones_parciales* a sus sub-problemas. El proceso se inicia cuando el *agente* encargado de la resolución del primer sub-problema comienza la búsqueda de la primera *solución_parcial*. Para ello, lo primero que hace es realizar una arco-consistencia direccional según el orden de ancho 1 d=$\{X_1,X_2,X_3,X_4,X_5,X_6\}$. Esto acota los dominios de las variables X_1 y X_4 como podemos ver en la Figura 5.11, generando el conjunto de soluciones que se muestran en la Figura 5.12.

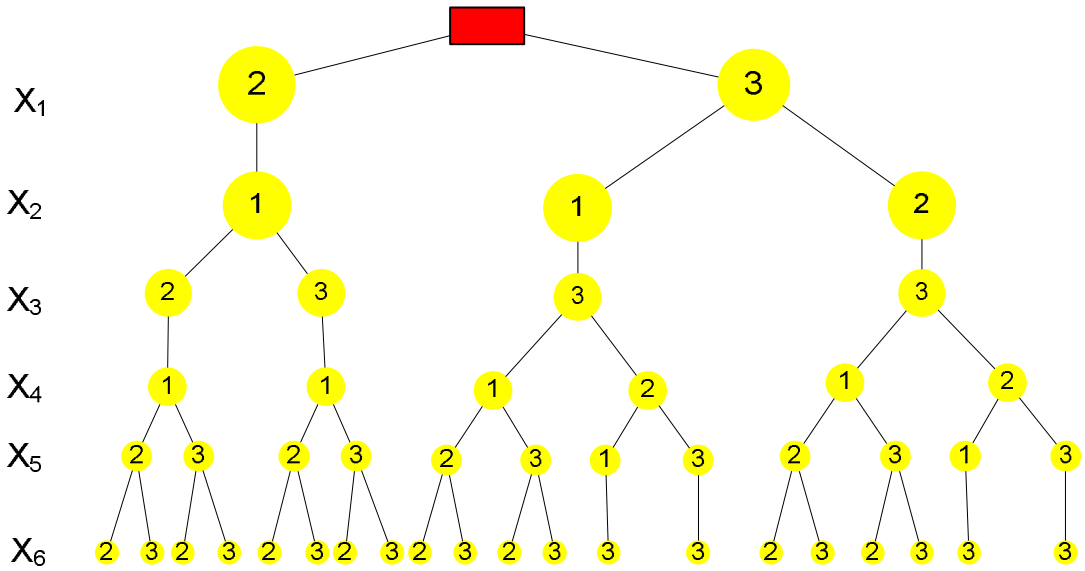


Figura 5.12: Conjunto de *soluciones_parciales* válidas, después de la arco-consistencia direccional.

Como, tras la arco-consistencia direccional, ningún dominio ha quedado vacío, TSA asigna valores a las variables y encuentra la primera *solución_parcial*: X_1=2, X_2=1,X_3=2,X_4=1,X_5=2,X_6=2. Esta solución no es consistente con el sub-problema 2, ya que los valores asignados a las variables X_2 y X_4 dejan vacío el dominio de X_7. Por lo tanto, se genera el mensaje *Nogood* (X_2=1, X_4=1). Cuando el *agente* encargado del sub-problema 1 recibe este

mensaje *Nogood*, marca el valor de X_4 acotado por X_2, lo que provoca la poda de 7 *soluciones_parciales* válidas para el sub-problema 1 pero inconsistentes con el problema global (ver Figura 5.13).

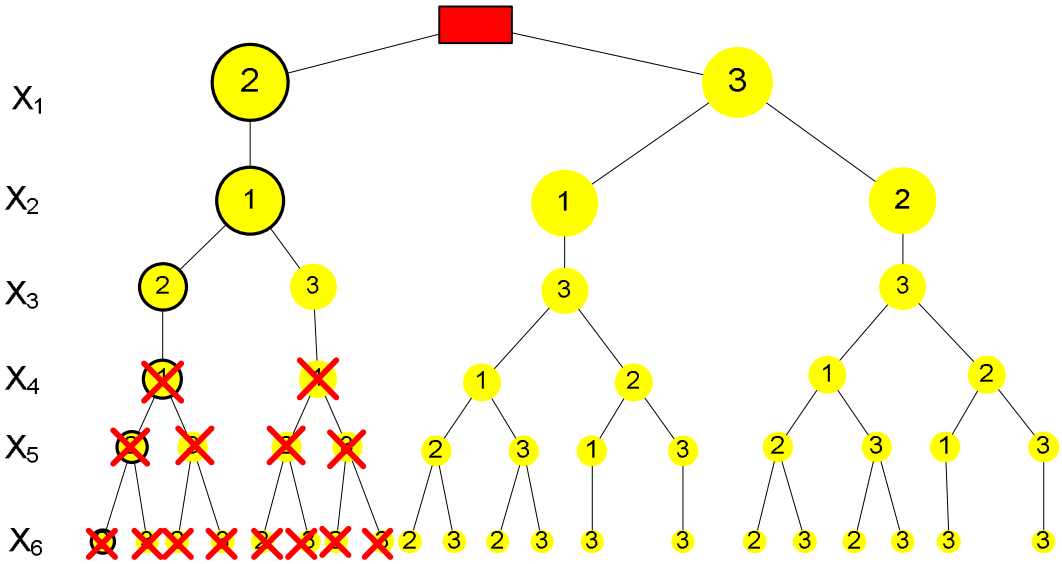


Figura 5.13: Poda del Algoritmo TSA tras recibir el *Nogood* (X_2=1,X_4=1).

A continuación el *agente* marca las variables X_4, X_2 y X_1 como *varNogood* y comienza a partir de X_4 a buscar una variable (marcada como *varNogood*), que tenga en su dominio valores disponibles para ser asignados. La primera variable que encuentra es X_1, así que comienza a reasignar valores a partir de esta variable. Esto le conduce a la solución: X_1=3, X_2=1,X_3=3,X_4=1,X_5=2,X_6=2 (Figura 5.14). Sin embargo, esta solución no cambia ninguno de los valores de las variables incluidas en el último mensaje *Nogood*, por lo que se repite el mismo proceso: se marca el valor de X_4 acotado por X_2, lo que en esta ocasión provoca la poda de 3 *soluciones_parciales* (ver Figura 5.14), y se marcan las variables X_4, X_2 y X_1 como *varNogood*. Ahora es X_4 la primera variable marcada como *varNogood* que tiene valores disponibles. Así que se reasignan valores a partir de X_4 y se obtiene la solución: X_1=3, X_2=1,X_3=3,X_4=2,X_5=1,X_6=3.

De nuevo se genera un mensaje *Nogood* (X_5=1, X_6=3), ya que la última *solución_parcial* del sub-problema 1 deja el dominio de la variable X_9 vacío.

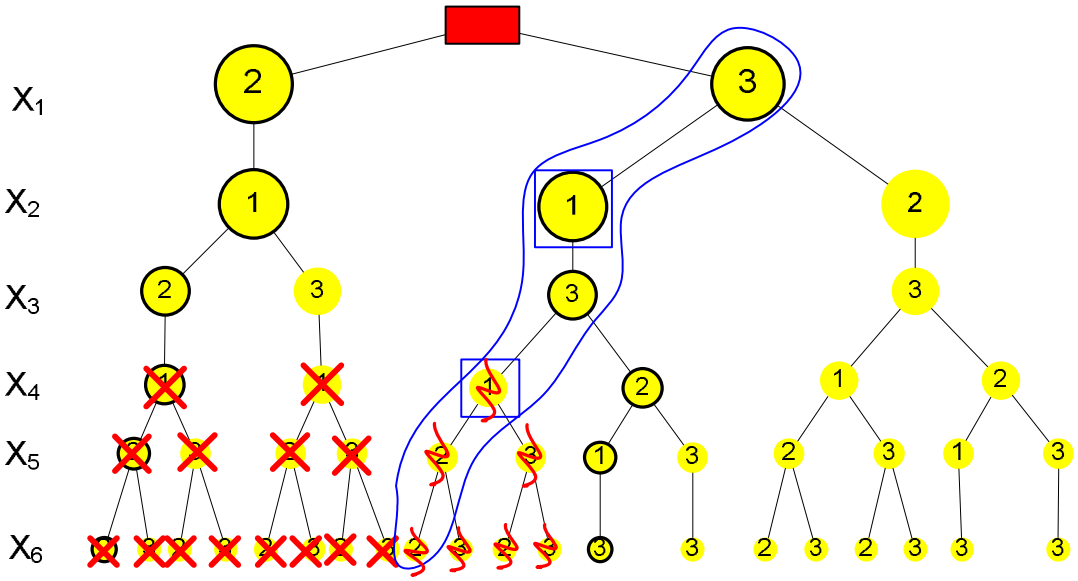


Figura 5.14: Segunda poda del Algoritmo TSA motivada por el *Nogood* (X_2=1, X_4=1).

Cuando el *agente* del sub-problema 1 recibe este último mensaje *Nogood*, marca el dominio de X_6 acotado por X_5, marca las variables X_5, X_6 y todos sus antecesores como *varNogood*. A partir de X_6, la primera variable con valores disponibles marcada como *varNogood* es X_5. La reasignación de valores a partir de X_5 nos conduce a la solución: X_1=3, X_2=1,X_3=3,X_4=2,X_5=3,X_6=3 (ver Figura 5.15). Esta *solución_parcial* es consistente con el sub-problema 2, por ejemplo con la instanciación: (X_7=2, X_8=1, X_9=3, X_{10}=2). De este modo, una solución al problema global ha sido encontrada, y el algoritmo DTS termina.

5.5. Conclusiones

A lo largo de este capítulo hemos presentado una nueva técnica para la resolución de CSPs Distribuidos. Esta técnica se basa en el algoritmo distribuido DTS el cual lleva a acabo una búsqueda en profundidad a lo largo de la *Estructura DFS-Tree CSP*, presentada en este capítulo. Esta estructura está compuesta por un conjunto de nodos organizados en una estructura con

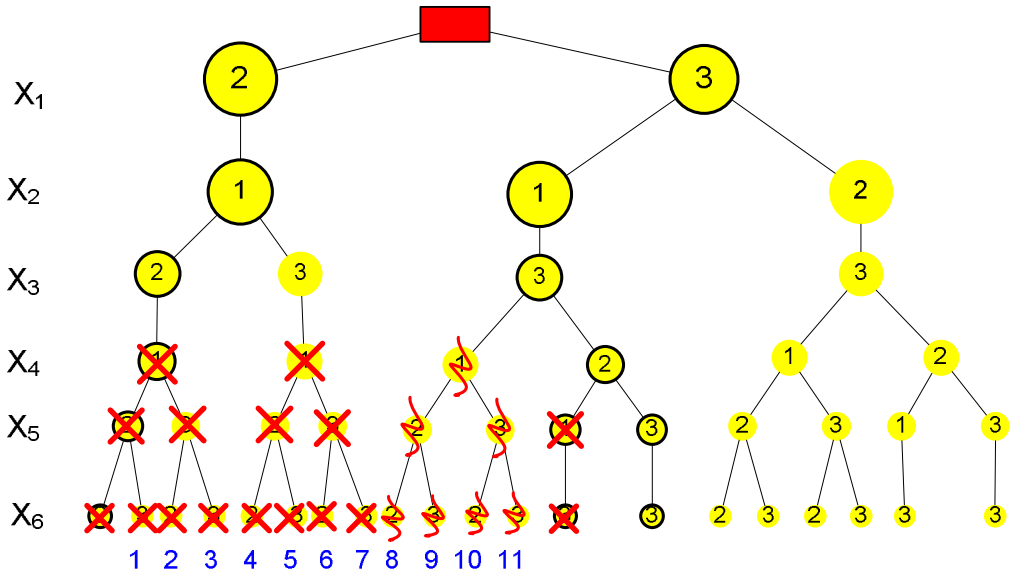


Figura 5.15: Poda del Algoritmo TSA tras recibir el *Nogood* (X_5=1,X_6=3).

forma de árbol y donde cada nodo representa un sub-problema que forma parte del problema global. Todo esto permite al algoritmo DTS la resolución de varios sub-problemas de manera asíncrona cuando recaen en diferentes ramas de la *Estructura DFS-Tree CSP*, ganando de este modo en eficiencia.

Además del algoritmo DTS, el cual coordina la resolución distribuida de todos los subproblemas, en este capítulo presentamos dos algoritmos (FCA y TSA) para realizar la búsqueda de *soluciones_parciales* de cada sub-problema. Estos algoritmos tienen la característica de podar el espacio de búsqueda de su sub-problema gracias a la información recibida con el paso de mensajes realizado por el algoritmo DTS y sin necesidad de almacenar estos mensajes, por lo que no tienen un gran coste espacial. En el capítulo 7 se llevará a cabo una evaluación empírica de los algoritmos presentados en este capítulo.

Capítulo 6

Búsqueda Distribuida heurística

En el apartado 5.4 hemos presentado dos algoritmos centralizados, FCA (Algoritmo 7) y TSA (Algoritmo 8), para resolver sub-CSPs haciendo uso de la información proporcionada por los mensajes *Nogood* que se generan durante el proceso de búsqueda distribuida del algoritmo DTS. En ambos algoritmos, fácilmente, se pueden incluir técnicas heurísticas que aprovechan la información del mensaje *Nogood* para podar en mayor medida el espacio de búsqueda. La poda llevada a cabo por estos algoritmos no garantiza la completitud por lo que algunas soluciones válidas pueden ser eliminadas durante el proceso de búsqueda.

6.1. Búsqueda intra-agente heurística: FCAH

En este apartado transformaremos el algoritmo FCA (Algoritmo 7) en el algoritmo Forward Checking Acotado Heurístico (FCAH), de manera que aproveche en mayor medida la información contenida en los mensajes *Nogood* para podar el espacio de búsqueda.

El algoritmo FCA utilizaba la información del mensaje *Nogood* de dos maneras dependientes de las características del mensaje:

1. Si el mensaje *Nogood* sólo contiene una variable del sub-problema, su valor será eliminado del dominio de la variable, por lo que se reanudará la búsqueda de una nueva solución a partir de esta variable tal y como lo

haría el algoritmo FC.

2. Si el mensaje *Nogood* contiene más de una variable pertenecientes al subproblema, el algoritmo FCA sólo tiene en cuenta las 2 variables de mayor orden, según la secuencia de búsqueda d, contenidas en el mensaje *Nogood* para marcar el dominio de la variable de mayor orden (X_j) acotado por la segunda variable de mayor orden (X_s). El resto de variables contenidas en el mensaje *Nogood* sólo se tienen en cuenta para asegurar que, en la nueva solución, ha cambiado el valor de alguna de las variables contenidas en el mensaje *Nogood*.

Cuando tenemos un mensaje *Nogood* de tipo 1, el algoritmo FCA ya utiliza al máximo la información proporcionada por el mensaje para podar el espacio de búsqueda. Sin embargo, cuando el mensaje *Nogood* es de tipo 2, para asegurar la completitud, el algoritmo FCA solamente poda el espacio de búsqueda dependiente de las dos variables de mayor orden contenidas en el mensaje *Nogood*. La técnica heurística que proponemos en este apartado se basa en utilizar más ampliamente la información de los mensaje *Nogood* de tipo 2 a costa de la posible pérdida de soluciones válidas.

La idea principal de la técnica heurística presentada en este apartado, es ir directamente a cambiar uno de los valores de las variables incluidas en el mensaje *Nogood*, podando el espacio de búsqueda relativo a los valores, todavía no asignados, de las variables situadas entre dos variables del mensaje *Nogood*. De este modo, si la variable (X_j) de mayor orden, contenida en el mensaje *Nogood*, hubiese agotado todos los valores de su dominio, FCA no haría backtracking a su variable predecesora (X_{j-1}), sino que, como ya hemos comentado, haría backtracking a la segunda variable (X_s) de mayor orden contenida en el mensaje *Nogood*. Este tipo de backtracking supone una posible pérdida de soluciones válidas, ya que estamos descartando valores, de las variables contenidas en el mensaje *Nogood*, que están acotados por los valores de otras variables no contenidas en el mensaje *Nogood*.

El Algoritmo 12 muestra el pseudocódigo del algoritmo FCAH. Se puede

observar que el algoritmo FCAH se comporta igual que el algoritmo FCA (Algoritmo 7) cuando se trata de buscar la primera *solución_parcial* del subproblema. Sin embargo para encontrar una nueva solución después de recibir un mensaje *Nogood* el algoritmo FCAH llama a la función *FCNogood* (en lugar de FC), que es una nueva versión del algoritmo FC, donde en el proceso de backtracking no salta a la variable anterior, según la secuencia de búsqueda, sino a la variable anterior, según d, que está contenida en el mensaje *Nogood* y que tiene valores disponibles.

Algoritmo FCAH(G, *mensaje_nogood*)
;G es el grafo de restricciones e incluye la secuencia de búsqueda d=(X_1,...,X_n)
begin
  if *primera_solucion* *;buscamos la primera solución*
    solucion=FC(X_1); *;aplicar algoritmo Forward Checking*
  else
    while (solucion $\neq$ NO_SOLUCION $\wedge$ ninguna variable
        incluida en el *mensaje_nogood* cambie su valor)
      if al menos 2 de mis variables pertenecen al *mensaje_nogood*
        marcar valor(X_j) acotado por X_s;
      ;X_j es la variable de mayor orden incluida en mensaje_nogood
      ;X_s es la 2^a variable de mayor orden incluida en mensaje_nogood
      solucion=FCNogood(G, *mensaje_nogood*);
  return solucion;
end FCAH

Algoritmo 12. Algoritmo Forward Checking Acotado Heurístico. Parte I.

FCNogood (Algoritmo 13) se comporta de manera similar al algoritmo FC cuando el mensaje *Nogood* sólo contiene una variable. En esta situación, la asignación de valores se lleva a cabo del mismo modo, realizando el mismo tipo de chequeo hacia delante (checkForward) cuando se asigna un nuevo valor, y en

caso de necesitar hacer un backtracking (porque el dominio de alguna variable se ha quedado sin valores disponibles), este se realiza en orden decreciente según la secuencia d.

```
procedure FCNogood(G, mensaje_nogood)
begin
    solucion=NO_SOLUCION;
    while solucion==NO_SOLUCION do
        k=j;
        ;X_j es la variable de mayor orden, según d, incluida en mensaje_nogood
        while D_k==Ø ∨ (X_k no pertenece V_nogood ∧
                        ∃ X_m / (m < k ∧ X_m pertenece V_nogood))
            k = k-1;
        if k < 1
            return NO_SOLUCION;
        for each X_i / (k ≤ i < n) ∧ X_i tiene valor asignado
            retractar los efectos de la asignación X_i=valor(X_i);
        solucion=asignarValor(k);
    return solucion;
end FCNogood
```

Algoritmo 13. Algoritmo Forward Checking Acotado Heurístico. Parte II.

```
procedure asignarValor(var)
begin
    solucion[X_var]= siguiente valor disponible en D_var;
    if var < n
        checkForward(X_var);
        ;Eliminar de los dominios de las variables, aún no
        ;instanciadas con un valor, aquellos valores inconsistente con respecto
        ;al valor asignado a la X_var, de acuerdo al conjunto de restricciones.
        if ningún dominio está vacío
            solucion=asignarValor(var + 1);
        else      ;algún dominio quedó vacío
            return NO_SOLUCION;
    return solucion;
end asignarValor
```

Algoritmo 14. Algoritmo Forward Checking Acotado Heurístico. Parte III.

A continuación mostraremos con dos ejemplos, la ejecución del algoritmo FCAH frente al algoritmo FCA. Utilizaremos los dos CSPs distribuidos representados en la Figura 6.1.

En la Figura 6.2 podemos ver el árbol de búsqueda (d=$\{X_1,\ X_2,\ X_3,\ X_4\}$) que genera el algoritmo FCA (Algoritmo 7), hasta encontrar una *solución_parcial* para el sub-problema Sub-PA1 consistente con el Sub-PA2 y por lo tanto, con el problema global CSP A. Vemos, que debido a la naturaleza de los mensajes *Nogood*, el algoritmo FCA no puede realizar ningún tipo de poda, por lo que se van generando todas las *soluciones_parciales* posibles según la secuencia de búsqueda seleccionada (X_1, X_2, X_3, X_4) y para cada una de ellas se recibe un mensaje *Nogood* que la invalida. Así continua el proceso hasta que se genera la solución (X_1=2, X_2=1, X_3=2, X_4=1) que es consistente con el sub-P2A. El proceso del algoritmo FCA se hace tan largo,

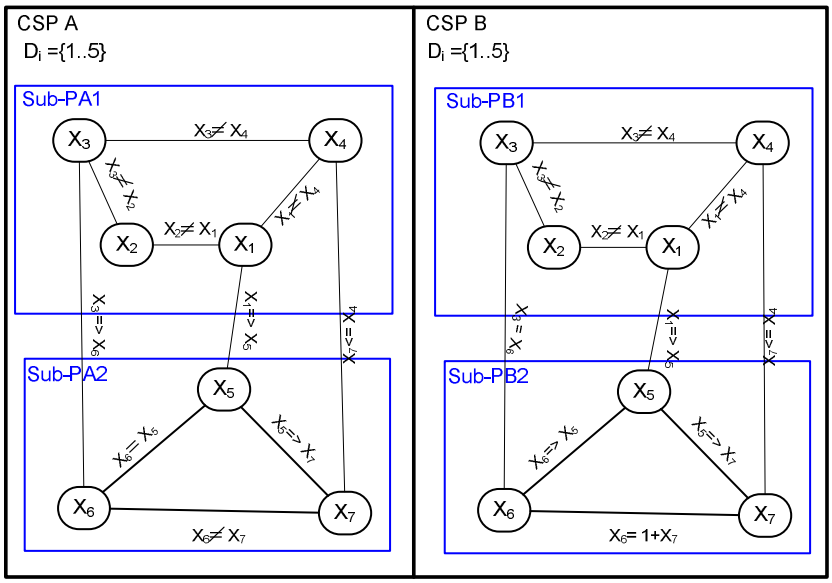
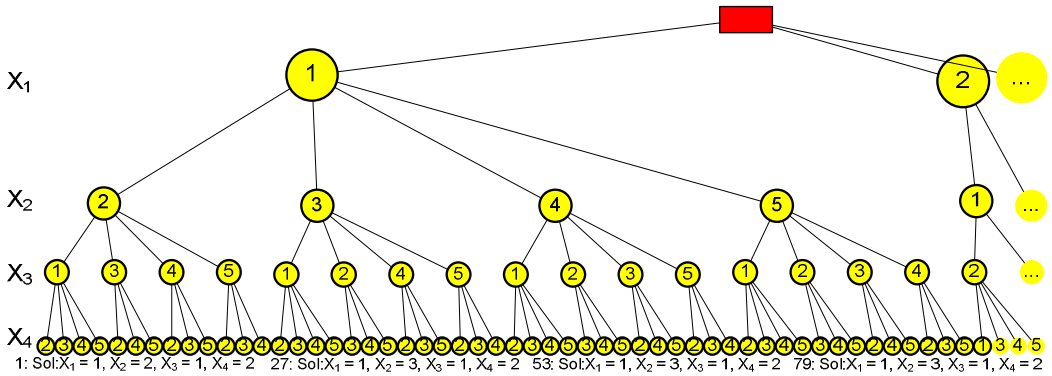


Figura 6.1: Ejemplos de CSPs Distribuidos.

para este ejemplo (Figura 6.1, CSP A), porque cuando se recibe un mensaje *Nogood* con la variable X_1, su valor no se tiene en cuenta, ya que en el mismo mensaje *Nogood* existen 2 variables de mayor orden (X_3 y X_4). Esto se hace para asegurar la completitud del algoritmo.

Por el contrario, el algoritmo FCAH sí que tiene en cuenta a la variable X_1 recibida en el mensaje *Nogood*. En la Figura 6.3 podemos ver el árbol de búsqueda (d=$\{X_1, X_2, X_3, X_4\}$) que genera el algoritmo FCAH (Algoritmo 12), hasta encontrar una *solución_parcial* para el sub-problema Sub-PA1 consistente con el Sub-PA2. Se observa como el algoritmo FCAH realiza una poda exhaustiva, saltando directamente a la variable X_1 cuando agota los valores de los dominios de las otras variables implicadas en el mensaje *Nogood* (X_3 y X_4), sin probar el resto de valores de la variable X_2, la cual no está implicada en ningún mensaje *Nogood*. Para este ejemplo (Figura 6.1, CSP A), el algoritmo FCAH ha descartado un gran número de *soluciones_parciales* que eran inconsistentes con el problema global, encontrando más rápidamente la primera *solución_parcial* válida para formar parte de la solución del problema global.



1: Sol:$X_1 = 1, X_2 = 2, X_3 = 1, X_4 = 2$
2: Nogood:$X_1 = 1, X_3 = 1, X_4 = 2$
3: Sol:$X_1 = 1, X_2 = 2, X_3 = 1, X_4 = 3$
4: Nogood:$X_1 = 1, X_3 = 1, X_4 = 3$
5: Sol:$X_1 = 1, X_2 = 2, X_3 = 1, X_4 = 4$
6: Nogood:$X_1 = 1, X_3 = 1, X_4 = 4$
7: Sol:$X_1 = 1, X_2 = 2, X_3 = 1, X_4 = 5$
8: Nogood:$X_1 = 1, X_3 = 1, X_4 = 5$
9: Sol:$X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 2$
10: Nogood:$X_1 = 1, X_3 = 3, X_4 = 2$
11: Sol:$X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 4$
12: Nogood:$X_1 = 1, X_3 = 3, X_4 = 4$
13: Sol:$X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 5$
14: Nogood:$X_1 = 1, X_3 = 3, X_4 = 5$
15: Sol:$X_1 = 1, X_2 = 2, X_3 = 4, X_4 = 2$
16: Nogood:$X_1 = 1, X_3 = 4, X_4 = 2$
17: Sol:$X_1 = 1, X_2 = 2, X_3 = 4, X_4 = 3$
18: Nogood:$X_1 = 1, X_3 = 4, X_4 = 3$
19: Sol:$X_1 = 1, X_2 = 2, X_3 = 4, X_4 = 5$
20: Nogood:$X_1 = 1, X_3 = 4, X_4 = 5$
21: Sol:$X_1 = 1, X_2 = 2, X_3 = 5, X_4 = 2$
22: Nogood:$X_1 = 1, X_3 = 5, X_4 = 2$
23: Sol:$X_1 = 1, X_2 = 2, X_3 = 5, X_4 = 3$
24: Nogood:$X_1 = 1, X_3 = 5, X_4 = 3$
25: Sol:$X_1 = 1, X_2 = 2, X_3 = 5, X_4 = 4$
26: Nogood:$X_1 = 1, X_3 = 5, X_4 = 4$
27: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 2$
28: Nogood:$X_1 = 1, X_3 = 1, X_4 = 2$
29: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 3$
30: Nogood:$X_1 = 1, X_3 = 1, X_4 = 3$
31: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 4$
32: Nogood:$X_1 = 1, X_3 = 1, X_4 = 4$
33: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 5$
34: Nogood:$X_1 = 1, X_3 = 1, X_4 = 5$
35: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 3$
36: Nogood:$X_1 = 1, X_3 = 2, X_4 = 3$
37: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 4$
38: Nogood:$X_1 = 1, X_3 = 2, X_4 = 4$
39: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 5$
40: Nogood:$X_1 = 1, X_3 = 2, X_4 = 5$
41: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 2$
42: Nogood:$X_1 = 1, X_3 = 4, X_4 = 2$
43: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 3$
44: Nogood:$X_1 = 1, X_3 = 4, X_4 = 3$
45: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 5$
46: Nogood:$X_1 = 1, X_3 = 4, X_4 = 5$
47: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 2$
48: Nogood:$X_1 = 1, X_3 = 5, X_4 = 2$
49: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 3$
50: Nogood:$X_1 = 1, X_3 = 5, X_4 = 3$
51: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 4$
52: Nogood:$X_1 = 1, X_3 = 5, X_4 = 4$
53: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 2$
54: Nogood:$X_1 = 1, X_3 = 1, X_4 = 2$
55: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 3$
56: Nogood:$X_1 = 1, X_3 = 1, X_4 = 3$
57: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 4$
58: Nogood:$X_1 = 1, X_3 = 1, X_4 = 4$
59: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 5$
60: Nogood:$X_1 = 1, X_3 = 1, X_4 = 5$
61: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 3$
62: Nogood:$X_1 = 1, X_3 = 2, X_4 = 3$
63: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 4$
64: Nogood:$X_1 = 1, X_3 = 2, X_4 = 4$
65: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 5$
66: Nogood:$X_1 = 1, X_3 = 2, X_4 = 5$
67: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 2$
68: Nogood:$X_1 = 1, X_3 = 4, X_4 = 2$
69: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 3$
70: Nogood:$X_1 = 1, X_3 = 4, X_4 = 3$
71: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 5$
72: Nogood:$X_1 = 1, X_3 = 4, X_4 = 5$
73: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 2$
74: Nogood:$X_1 = 1, X_3 = 5, X_4 = 2$
75: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 3$
76: Nogood:$X_1 = 1, X_3 = 5, X_4 = 3$
77: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 4$
78: Nogood:$X_1 = 1, X_3 = 5, X_4 = 4$
79: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 2$
80: Nogood:$X_1 = 1, X_3 = 1, X_4 = 2$
81: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 3$
82: Nogood:$X_1 = 1, X_3 = 1, X_4 = 3$
83: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 4$
84: Nogood:$X_1 = 1, X_3 = 1, X_4 = 4$
85: Sol:$X_1 = 1, X_2 = 3, X_3 = 1, X_4 = 5$
86: Nogood:$X_1 = 1, X_3 = 1, X_4 = 5$
87: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 3$
88: Nogood:$X_1 = 1, X_3 = 2, X_4 = 3$
89: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 4$
90: Nogood:$X_1 = 1, X_3 = 2, X_4 = 4$
91: Sol:$X_1 = 1, X_2 = 3, X_3 = 2, X_4 = 5$
92: Nogood:$X_1 = 1, X_3 = 2, X_4 = 5$
93: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 2$
94: Nogood:$X_1 = 1, X_3 = 4, X_4 = 2$
95: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 3$
96: Nogood:$X_1 = 1, X_3 = 4, X_4 = 3$
97: Sol:$X_1 = 1, X_2 = 3, X_3 = 4, X_4 = 5$
98: Nogood:$X_1 = 1, X_3 = 4, X_4 = 5$
99: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 2$
100: Nogood:$X_1 = 1, X_3 = 5, X_4 = 2$
101: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 3$
102: Nogood:$X_1 = 1, X_3 = 5, X_4 = 3$
103: Sol:$X_1 = 1, X_2 = 3, X_3 = 5, X_4 = 4$
104: Nogood:$X_1 = 1, X_3 = 5, X_4 = 4$
105: Sol:$X_1 = 2, X_2 = 1, X_3 = 2, X_4 = 1$

Figura 6.2: Árbol de búsqueda, soluciones y mensajes *Nogood* del Sub-PA1 (Figura 6.1, CSP A) siguiendo el Algoritmo FCA.

Sin embargo, como ya hemos comentado antes, el algoritmo FCAH no es completo y en su proceso de backtracking, puede perder *soluciones_parciales* que forman parte de una solución del problema global. Un ejemplo de poda con pérdida de *soluciones_parciales* válidas lo podemos ver en la resolución del CSP B de la Figura 6.1. En la Figura 6.4 podemos ver el árbol de búsqueda (d=$\{X_1, X_2, X_3, X_4\}$) generado por el algoritmo FCA hasta encontrar la primera *solución_parcial* del sub-problema Sub-PB1 (X_1=1, X_2=3, X_3=2, X_4=3) consistente con el problema global CSP B (ver Figura 6.1). Sin embargo, esta solución nunca sería encontrada por el algoritmo FCAH debido a su proceso

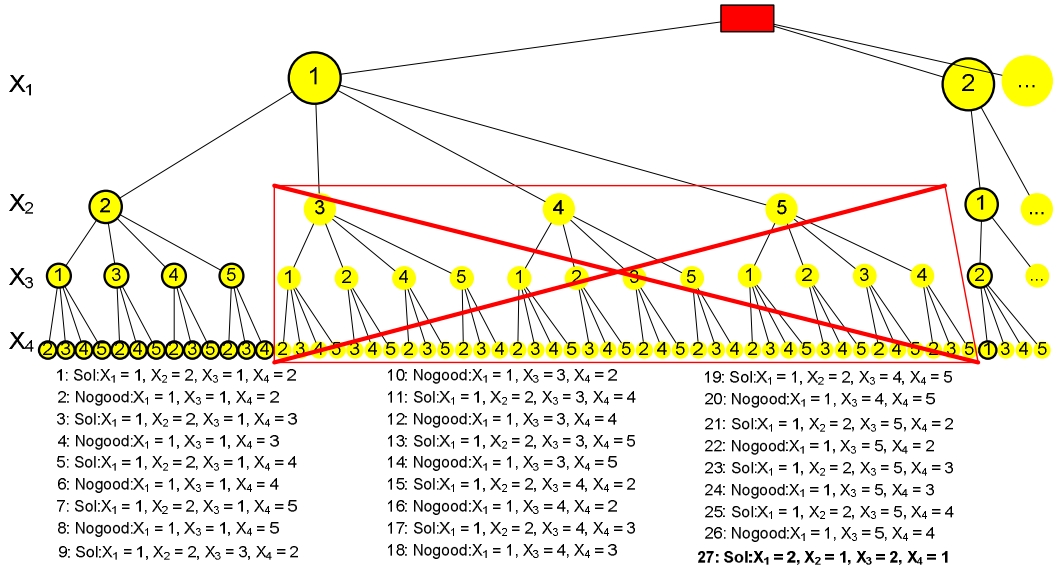


Figura 6.3: Árbol de búsqueda, soluciones y mensajes *Nogood* del Sub-PA1 (Figura 6.1, CSP A) siguiendo el Algoritmo FCAH.

de backtracking. En la Figura 6.5 podemos ver como FCAH poda esta solución debido al backtracking que realiza hasta la variable X_1 sin chequear todos los valores de la variable X_2. Aun a pesar de podar la primera *solución_parcial* válida, el algoritmo FCAH encuentra antes que el algoritmo FCA una *solución_parcial* consistente con el problema global. Sin embargo, en otros casos, la poda de una *solución_parcial* válida puede implicar que no se encuentre una solución al problema global, cuando realmente sí que la tiene. De esta forma, la heurística presentada es correcta pero no completa.

6.2. Búsqueda heurística en estructuras de árbol: TSAH

En este apartado modificaremos el algoritmo TSA (Algoritmo 8) para convertirlo en el algoritmo *Tree Search Acotado Heurístico* (TSAH). La heurística aplicada se encarga de guiar el proceso de backtracking del algoritmo. Cuando un nodo no tiene ninguna asignación válida para su variable, para que el

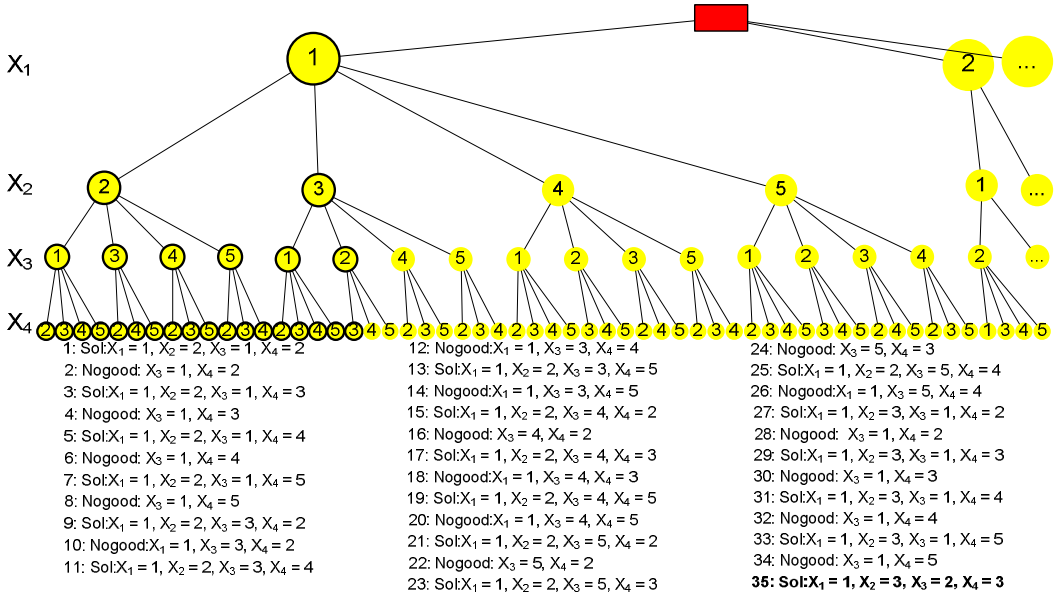


Figura 6.4: Árbol de búsqueda, soluciones y mensajes *Nogood* del Sub-PB1 (Figura 6.1, CSP B) siguiendo el Algoritmo FCA.

proceso de búsqueda continue, el algoritmo TSAH debe volver a alguno de los nodos previos para cambiar la asignación de su variable; es en este momento cuando aplicamos la heurística con el objetivo de decidir a qué nodo debe saltar el proceso de búsqueda. El algoritmo TSA realiza el proceso de backtracking saltando a la variable anterior más cercana, con valores disponibles en su dominio y marcada como *varNogood*. Esto significa que salta a la variable anterior más cercana entre:

- su variable padre (su antecesora más cercana);
- otra variable del actual mensaje *Nogood*;
- otra variable antecesora de alguna de las variables del actual mensaje *Nogood*;
- otra variable anteriormente involucrada en un mensaje *Nogood* y todavía no desmarcada como *varNogood*, es decir, todavía no se ha cambiado el valor de ninguna de sus variables antecesoras;

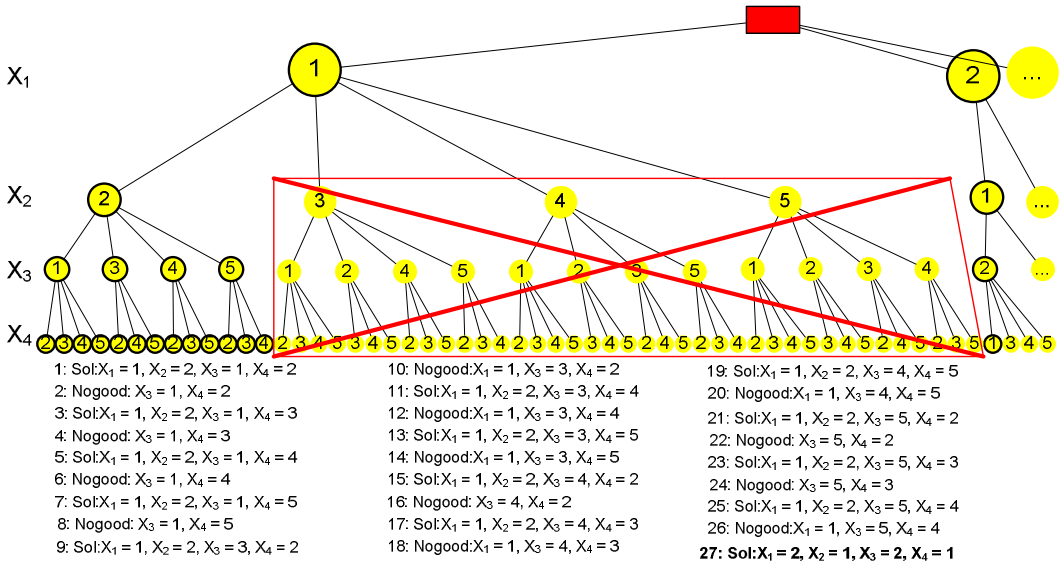


Figura 6.5: Árbol de búsqueda, soluciones y mensajes *Nogood* del Sub-PB1 (Figura 6.1, CSP B) siguiendo el Algoritmo FCAH.

- otra variable que tiene una variable sucesora anteriormente involucrada en un mensaje *Nogood* y todavía no está desmarcada como *varNogood*.

El algoritmo TSAH puede mantener varios niveles de completitud en función de la heurística aplicada, sin llegar a garantizar la completitud ninguno de ellos. A continuación presentamos varias heurísticas cuya diferencia radica en como se maneja la información recibida de los mensajes *Nogood*.

- **Heurística 1**: El algoritmo TSA, lo primero que hace cuando recibe un mensaje *Nogood* es marcar como *varNogood* a todas las variables incluidas en el mensaje *Nogood* y a todas sus respectivas antecesoras según la secuencia de búsqueda. Aquí radica la diferencia entre el algoritmo TSA y el algoritmo TSAH cuando se aplica la Heurística 1. Esta heurística solamente marca como *varNogood* a las variables incluidas en el mensaje *Nogood*, no a las antecesoras de estas variables. Así durante el proceso de backtracking TSAH con Heurística 1 salta a la anterior variable con valores disponibles en su dominio, seleccionando la más cercana entre:
  - su variable padre;

  - otra variable del actual mensaje *Nogood*;
  - otra variable anteriormente involucrada en un mensaje *Nogood* y todavía no desmarcada como *varNogood*.

- **Heurística 2**: En el algoritmo TSA, una variable solamente es desmarcada como *varNogood* cuando se cambia el valor asignado a su antecesora. Aquí radica el cambio del algoritmo TSA frente al TSAH cuando aplica la Heurística 2. En el algoritmo TSAH cada vez que se reciba un nuevo mensaje *Nogood*, todas las variables marcadas como *varNogood* serán desmarcadas y se marcarán de nuevo solamente las variables incluidas en el actual mensaje *Nogood*. Como se puede deducir, la Heurística 2 incluye a la Heurística 1 pero además la Heurística 2 no conserva ningún tipo de información de anteriores mensajes *Nogood*. Así durante el proceso de backtracking, TSAH con Heurística 2 salta a la anterior variable con valores disponibles en su dominio, seleccionando la más cercana entre:
  - su variable padre;
  - otra variable del actual mensaje *Nogood*.

- **Heurística 3**: La Heurística 3 acumula la Heurística 1 y la Heurística 2 y además no tiene en cuenta la estructura de árbol para guiar el proceso de backtracking. La Heurística 3 realiza el proceso de backtracking de manera similar al algoritmo FCAH (apartado 6.1). Así durante el proceso de backtracking, TSAH con Heurística 3 salta a la anterior variable incluida en el actual mensaje *Nogood* con valores disponibles en su dominio. En caso de que no exista una variable de menor orden incluida en el mensaje *Nogood*, entonces hace el backtracking a su variable padre.

A continuación se muestran las modificaciones que hay que realizar en el pseudocódigo del algoritmo TSA para transformarlo en el algoritmo TSAH según la heurística que se aplique. El pseudocódigo para el algoritmo TSAH es idéntico al pseudocódigo del algoritmo TSA (Algoritmos 8, 9 y 10) realizando unos pequeños cambios.

Para que el algoritmo TSAH utilice la Heurística 1 debemos cambiar en el pseudocódigo del Algoritmo 9 línea 1. la función *marcarVariablesNogood* por la nueva función *marcarVarNogoodHeuristico* descrita en el Algoritmo 15. Este cambio evitará que se marquen como *varNogood* las variables antecesoras de las variables incluidas en el mensaje *Nogood*. También se debe cambiar la línea 12 del Algoritmo 9 por: $k = max(padre(X_k), antVarNogood)$;

```
procedure marcarVarNogoodHeuristico(ConjVar, mensaje_nogood)
begin
    for each X_i / (X_i ∈ ConjVar ∧ X_i ∈ mensaje_nogood ) do
        marcar X_i como varNogood;
end marcarVarNogoodHeuristico
```

Algoritmo 15. Nueva función para marcar como *varNogood* las variables incluidas en los mensajes *Nogood*.

Para que el algoritmo TSAH utilice la Heurística 2, además de los cambios ya realizados para utilizar la Heurística 1, debemos añadir en el pseudocódigo del Algoritmo 9 entre las línea 1 y 2 la función *desmarcarVarNogood*(V_{agente_id}) mostrada en el Algoritmo 16. Con este cambio evitaremos mantener la información relativa a las variables incluidas en anteriores mensajes *Nogood*.

```
procedure desmarcarVarNogood(ConjVar)
begin
    for each X_i / (X_i ∈ ConjVar ∧ X_i marcada como varNogood) do
        desmarcar X_i como varNogood;
end desmarcarVarNogood
```

Algoritmo 16. Nueva función para desmarcar las variables marcadas como *varNogood*.

Para que el algoritmo TSAH utilice la Heurística 3, debemos mantener los cambios realizados para las Heurísticas 1 y 2, pero en este caso la línea 12 del

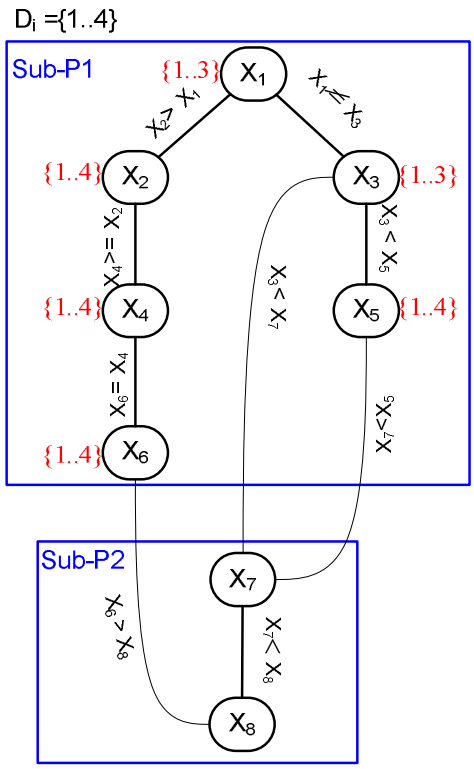


Figura 6.6: Ejemplo de CSP Distribuido.

Algoritmo 9 debe ser sustituida por:

```
12.  if existe antVarNogood
13.      k=antVarNogood;
14.  else
15.      k=padre(X_k);
```

La Figura 6.6 nos muestra un ejemplo de CSP Distribuido. Utilizaremos este CSP Distribuido para comprobar las diferencias entre la ejecución del algoritmo TSA y el algoritmo TSAH utilizando las diferentes heurísticas presentadas en este apartado.

En la Figura 6.7 podemos ver el árbol de búsqueda que genera el algoritmo TSA (Algoritmo 8), siguiendo la secuencia: d=$\{X_1, X_2, X_3, X_4, X_5, X_6\}$,

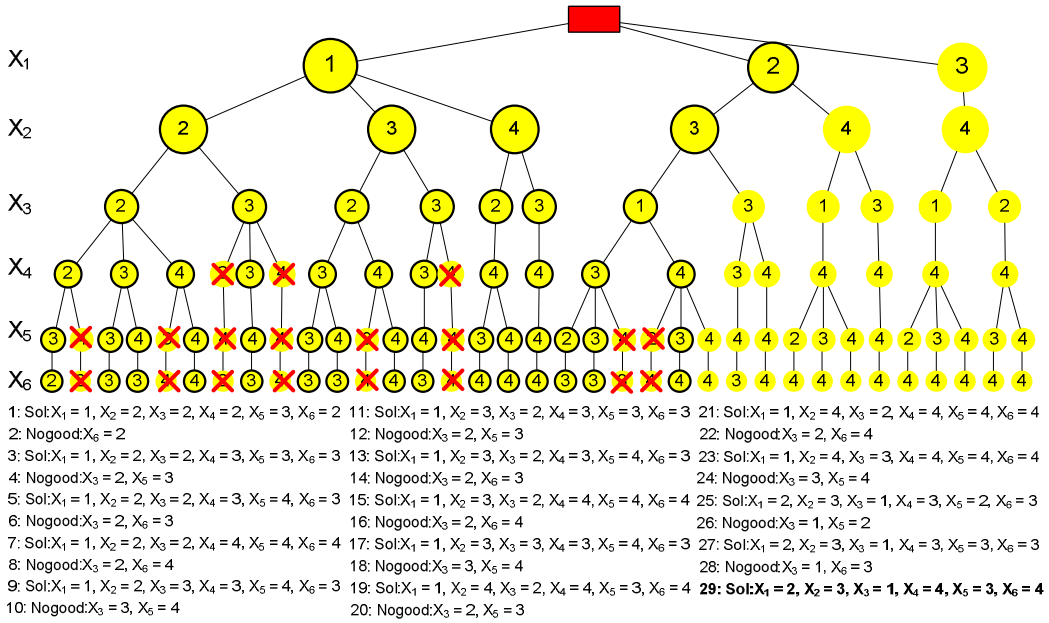


Figura 6.7: Árbol de búsqueda, soluciones y mensajes *Nogood* del Sub-P1 (Figura 6.6) siguiendo el Algoritmo TSA.

hasta encontrar una *solución_parcial* para el sub-problema Sub-P1 consistente con el problema global: X_1=2, X_2=3, X_3=1, X_4=4, X_5=3, X_6=4. Vemos, que el algoritmo TSA genera 14 *soluciones_parciales* inconsistentes con el problema global (Figura 6.6) antes de encontrar una *solución_parcial* consistente con el Sub-P2. Durante el proceso de búsqueda, el algoritmo TSA poda 8 *soluciones_parciales* gracias a la información recibida en los mensajes *Nogood*. Sin embargo, ninguna de ellas era una *solución_parcial* consistente con el problema global.

En el ejemplo de CSP Distribuido representado en la Figura 6.6, el árbol de búsqueda del algoritmo TSAH cuando utiliza la Heurística 1 y la Heurística 3 es el mismo. Podemos verlo en la Figura 6.8. En este caso se generan 8 *soluciones_parciales* del sub-P1 inconsistentes con el problema global (Figura 6.6) antes de encontrar la *solución_parcial* válida a nivel global: X_1=3, X_2=4, X_3=1, X_4=4, X_5=3, X_6=4. Vemos que la *solución_parcial* encontrada no coincide con la solución encontrada por el algoritmo TSA. Esto se debe a

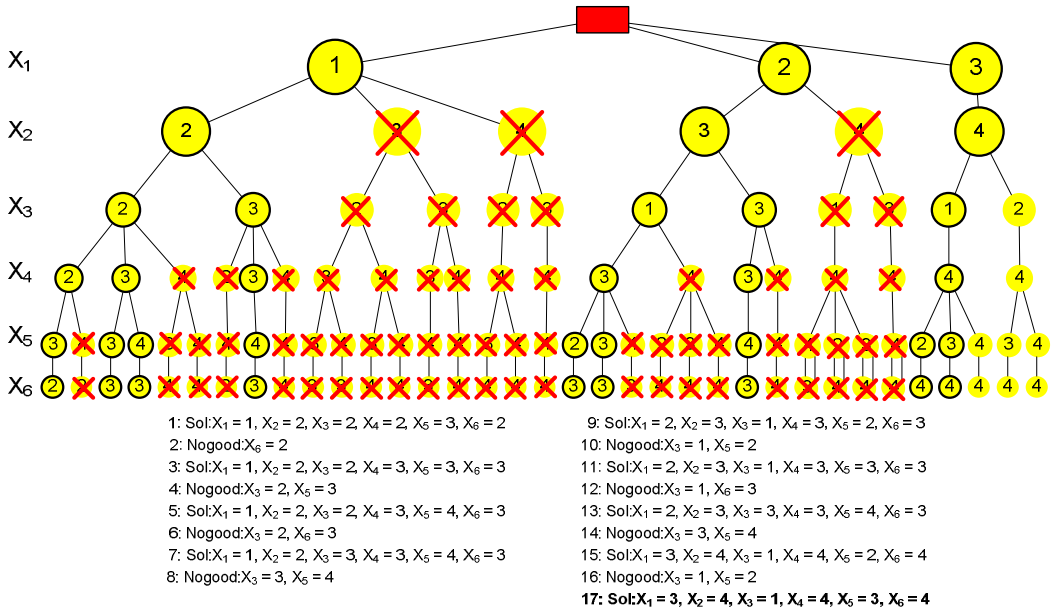


Figura 6.8: Árbol de búsqueda, soluciones y mensajes *Nogood* del Sub-P1 (Figura 6.6) siguiendo el Algoritmo TSAH, Heurística 1 o Heurística 3.

que durante el proceso de búsqueda, tanto la Heurística 1 como la Heurística 3 han podado la primera *solución_parcial* que era válida a nivel global, y que es la que proporcionaba el algoritmo TSA. Sin embargo, gracias a la poda realizada, el proceso de comunicación entre los agentes encargados de resolver los sub-problemas sub-P1 y sub-P2 se ha visto reducido, ya que para encontrar una solución global se han tenido que comunicar 6 *soluciones_parciales* menos usando las heurísticas que usando el algoritmo completo TSA.

En la Figura 6.9 podemos ver el árbol de búsqueda generado por el algoritmo TSAH cuando utiliza la Heurística 2 para resolver el CSP Distribuido representado en la Figura 6.6. Podemos observar que para este ejemplo en concreto, es la Heurística 2 la que mejor comportamiento tiene, ya que consigue encontrar una solución válida comunicando solamente 7 *soluciones_parciales* inconsistentes. Además, durante el proceso de búsqueda no ha podado ninguna *solución_parcial* válida, por lo que finalmente encuentra la misma solución que el algoritmo TSA.

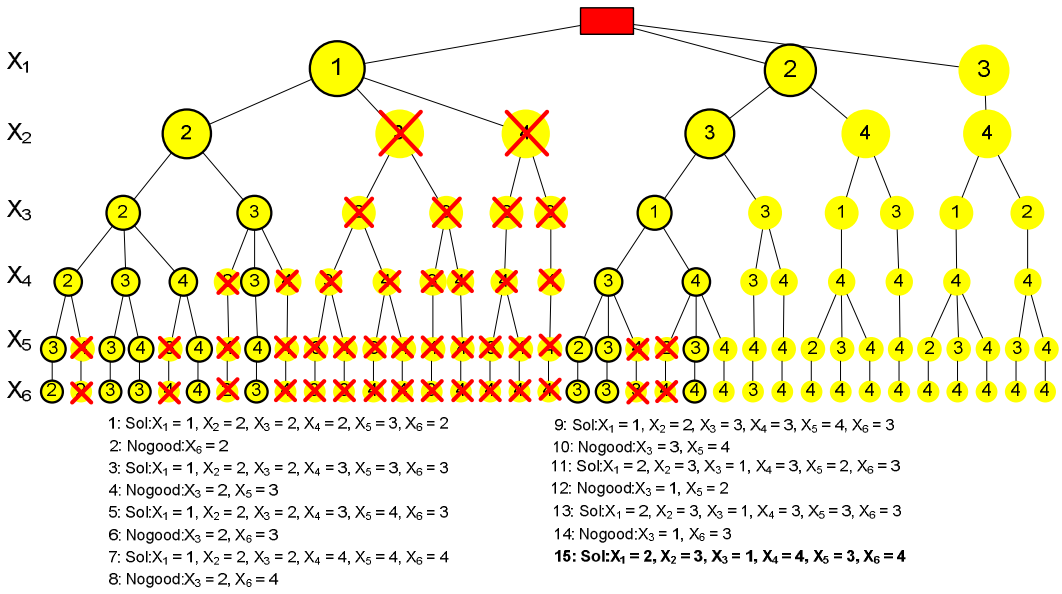


Figura 6.9: Árbol de búsqueda, soluciones y mensajes *Nogood* del Sub-P1 (Figura 6.6) siguiendo el Algoritmo TSAH, Heurística 2.

6.3. Conclusiones

En este capítulo hemos presentado varias técnicas heurísticas no completas para llevar a cabo la búsqueda de *soluciones_parciales* de cada uno de los sub-problemas que constituyen un problema distribuido. Concretamente, se ha presentado una nueva técnica heurística (Algoritmos 12, 13 y 14) la cual está basada en el algoritmo FCA, pero realiza una poda más drástica del espacio de soluciones del sub-problema, acorde a la información recibida en los mensajes *Nogood*. Además se han presentados otras tres técnicas heurísticas basadas en el algoritmo TSA, las cuales tienen diferentes niveles de completitud en función del uso que realizan de los mensajes *Nogood*.

Capítulo 7

Evaluación DCSP

Para evaluar el comportamiento de los algoritmos presentados en esta tesis doctoral vamos a realizar una evaluación empírica sobre varios tipos de problemas.

Una parte importante de la evaluación se realizará sobre problemas generados aleatoriamente. Algunos de ellos con un diseño centralizado, en los que no se pretende crear clusters artificiales, sino simular problemas que por naturaleza son centralizados. Otro conjunto distinto de problemas generados aleatoriamente tendrán un diseño distribuido, en los que, a priori, crearemos un conjunto de clusters semi-independientes para simular problemas que por naturaleza son distribuidos.

Otra parte importante de la evaluación empírica se llevará a cabo sobre el problema de planificación ferroviaria. Este es un problema que actualmente se resuelve de manera centralizada. Sin embargo, debido a la desregularización de los operadores ferroviarios, varios operadores deben competir por el uso de una misma infraestructura ferroviaria, lo que implica una mayor necesidad en la mejora del uso de la capacidad ferroviaria, una mejora en la gestión de los datos estratégicos (privados) de las diferentes operadoras ferroviarias y la necesidad de planificar trenes con trayectos internacionales. Todo esto nos ha conducido a seleccionar este tipo de problema como un buen candidato para evaluar la aplicabilidad de nuestras propuestas de particionamiento sobre un problema a priori centralizado.

En la evaluación llevamos a cabo una comparativa entre las diferentes propuestas de particionamiento de problemas: particionamiento de grafos, identificación de árboles, identificación de entidades. Tras realizar el particionamiento del problema, la versión distribuida del problema será resuelto con el algoritmo distribuido propuesto en esta tesis: DTS, cuyo comportamiento será comparado con la resolución del problema (sin distribuir) con diferentes algoritmos centralizados.

7.1. Problemas aleatorios

En esta sección llevamos a cabo una evaluación empírica, sobre problemas generados aleatoriamente, de las diferentes técnicas de particionamiento de CSPs, resueltas con el algoritmo distribuido DTS, y comparadas con la resolución centralizada de los mismos problemas.

7.1.1. Introducción

A continuación presentamos los resultados obtenidos en la evaluación empírica realizada entre DTS y varios resolvedores de CSP centralizados. Esta evaluación se ha llevado a cabo sobre dos tipos diferentes de problemas aleatorios: problemas aleatorios centralizados y problemas aleatorios distribuidos.

Los problemas aleatorios centralizados representan un conjunto de problemas en los que no se introduce ningún cluster artificialmente, por lo tanto si surge algún tipo de cluster es fuente de la aleatoriedad. Este tipo de problemas es definido por la 4-tupla $< n, d, a, p >$, donde n indica el número de variables, d la talla del dominio de las variables, a la aridad de restricciones binarias y p es un parámetro únicamente utilizado por el modelo distribuido que indica el número de particiones en las que se tiene que dividir el problema. Los problemas son generados aleatoriamente modificando estos parámetros.

Los problemas aleatorios distribuidos representan un conjunto de problemas en los que se fuerza cierta clusterización para simular un problema inherentemente distribuido. Este tipo de problema es definido por la 8-tupla $< a, n, k, t, p1, p2, q1, q2 >$, donde a indica el número de clusters que se deben generar, n es el número de variables de cada cluster, k es la talla del dominio de las variables, t representa la probabilidad de que dos clusters estén conectados, $p1$ es el porcentaje de *inter-restricciones* que existen entre dos clusters conectados, $p2$ es el porcentaje de *intra-restricciones* existentes en cada clusters, $q1$ es la restringibilidad de cada *inter-restricción* (probabilidad de que un par de valores esté prohibido en una *inter-restricción*) y $q2$ es la restringibilidad de cada *intra-restricción* (probabilidad de que un par de valores esté prohibido en una *intra-restricción*). Este tipo de parámetros son usados en evaluaciones experimentales de algoritmos para CSPs distribuidos [36]. Los problemas se generan aleatoriamente modificando estos parámetros.

7.1.2. Particionamiento de grafos

En este apartado estudiaremos el comportamiento del algoritmo distribuido DTS cuando realizamos una distribución de un problema usando el software de particionamiento de grafos METIS. Los problemas usados para realizar la evaluación son problemas generados aleatoriamente con un diseño centralizado. Los resultados obtenidos por el algoritmo distribuido son comparados con los tiempos de ejecución obtenidos por un bien conocido algoritmo de resolución de CSPs centralizado: Forward Checking (FC)[1].

En las Tablas 7.1 y 7.2 comparamos los tiempos de ejecución (en segundos) del modelo distribuido por METIS y resuelto con el algoritmo DTS con los tiempos de ejecución del mismo problema resuelto de manera centralizada. En la Tabla 7.1, se ha fijado el dominio de las variables, la aridad de las

[1]Forward Checking fue obtenido de CON'FLEX. Este software puede ser encontrado en: http://www-bia.inra.fr/T/conflex/ Logiciels/adressesConflex.html.

restricciones binarias y la talla de la partición, y el número de variables se incrementó desde 50 hasta 500. Se puede observar que los tiempos de ejecución de los problemas pequeños fueron peor en el caso del modelo distribuido que los del modelo centralizado. Sin embargo, cuando el número de variables incrementa, el comportamiento del modelo distribuido es mejor. En la Tabla 7.2, el número de variables, el dominio de las variables y la aridad de las restricciones binarias es fijo, en esta evaluación hemos incrementado la talla de la partición desde 2 hasta 20. De este modo podemos observar la importancia de la talla de la partición en los modelos distribuidos. Para los problemas pequeños, el número de particiones debe ser pequeño. Sin embargo, para grandes CSPs la talla de la partición debe ser más grande. En este caso, el número apropiado de particiones es 6.

Tabla 7.1: Instancias aleatorias $< n, d, a, p >$, n: variables, d: talla dominio, a: aridad y p: talla partición.

Problema	*Modelo Distribuido*	*Modelo Centralizado*
$< 50, 10, 25, 10 >$	12	3
$< 100, 10, 25, 10 >$	12	14
$< 150, 10, 25, 10 >$	15	37
$< 200, 10, 25, 10 >$	16	75
$< 250, 10, 25, 10 >$	17	98
$< 300, 10, 25, 10 >$	19	140
$< 350, 10, 25, 10 >$	23	217
$< 400, 10, 25, 10 >$	30	327
$< 450, 10, 25, 10 >$	32	440
$< 500, 10, 25, 10 >$	42	532

7.1.3. Identificación de estructuras de árbol

En este apartado realizamos una evaluación entre DTS, usando el algoritmo *ParticionarArboles* (Algoritmo 1) para realizar una distribución basada en árboles, y un bien conocido algoritmo centralizado: Forward Checking (FC)[2].

[2]FC fue obtenido de: http://ai.uwaterloo.ca/ vanbeek/software/software.html

Tabla 7.2: Instancias aleatorias $< n, d, a, p >$, n: variables, d: talla dominio, a: aridad y p: talla partición.

Problema	*Modelo Distribuido*	*Modelo Centralizado*
$< 200, 10, 25, 2 >$	51	75
$< 200, 10, 25, 3 >$	26	75
$< 200, 10, 25, 4 >$	20	75
$< 200, 10, 25, 5 >$	19	75
$< 200, 10, 25, 6 >$	13	75
$< 200, 10, 25, 7 >$	14	75
$< 200, 10, 25, 8 >$	15	75
$< 200, 10, 25, 9 >$	16	75
$< 200, 10, 25, 15 >$	18	75
$< 200, 10, 25, 20 >$	22	75

La evaluación en este caso se realiza sobre problemas aleatorios con diseño distribuido los cuales siguen la 8-tupla $< a, n, k, t, p1, p2, q1, q2 >$. Los resultados muestran los valores medios de 50 problemas aleatorios de cada instancia. Cada problema fue ejecutado con un tiempo límite de ejecución (TIME_OUT) de 900 segundos.

En las Figuras 7.1 y 7.2 se comparan los tiempos de ejecución de los algoritmos DTS y FC. En la Figura 7.1, el número de variables de cada sub-CSP es incrementado desde 10 hasta 40, el resto de parámetros permanecen fijos según la tupla ($< 5, n, 8, 0.2, 0.01, 0.2, 0.9, 0.3 >$). Se puede observar que el algoritmo distribuido DTS tiene un mejor comportamiento que el algoritmo centralizado FC en todas las instancias del problema. Además, DTS tiene menos ejecuciones que alcanzan el TIME_OUT que FC. A medida que el número de variables aumenta, también aumenta el número de ejecuciones de instancias del problema que alcanzan el TIME_OUT sin ser resueltas. Este es el motivo por el que en la gráfica se observa que el tiempo de ejecución medio de las instancias del problema va aumentando cada vez más despacio y nunca alcanza los 900 segundos (TIME_OUT).

En la Figura 7.2, la talla del dominio de las variables es aumentada desde 5 hasta 40, el resto de parámetros permanece fijo de acuerdo a la tupla ($<$

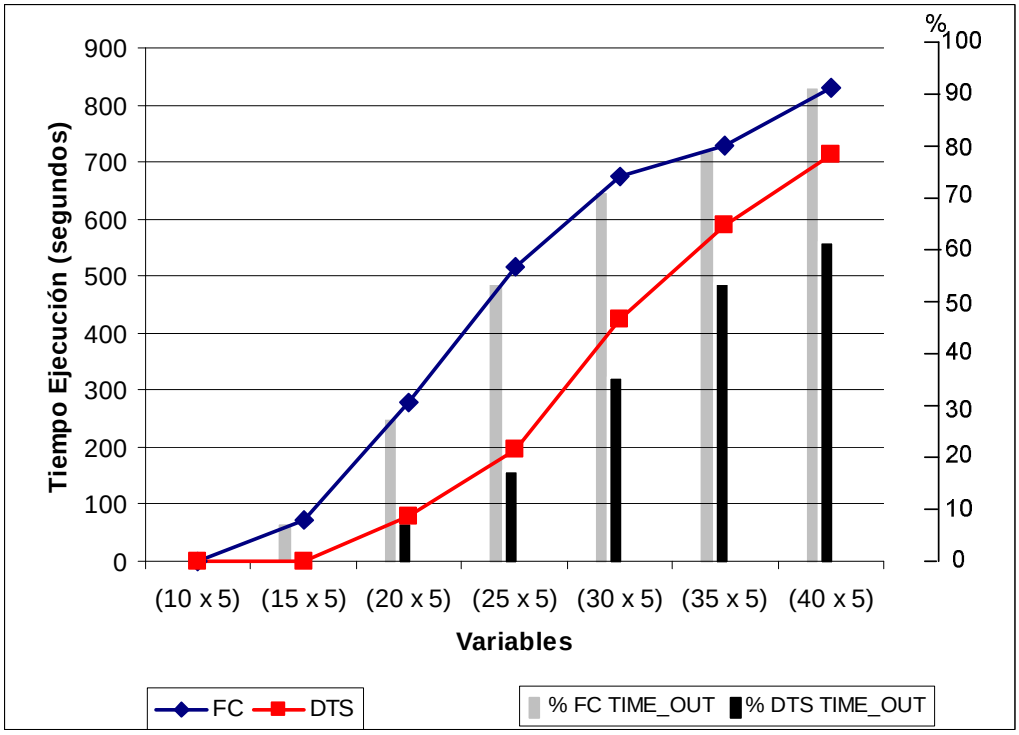


Figura 7.1: Tiempos de ejecución y porcentaje de soluciones que superan el $TIME\_OUT$ en problemas con diferente número de variables.

$5, 15, d, 0.2, 0.01, 0.2, 0.9, 0.3 >$). Se puede observar que el algoritmo distribuido DTS, de nuevo, obtiene mejores resultados que el algoritmo centralizado FC. A medida que aumenta el tamaño del dominio de las variables también aumenta el coste computacional de ambos algoritmos y un mayor número de ejecuciones del problema acaban sin ser resueltas debido a que alcanzan el TIME_OUT. Sin embargo, de nuevo, DTS tiene menor número de TIME_OUTs que FC.

La Figura 7.3 muestra el comportamiento de DTS y FC cuando se incrementan el valor de $q2$ en varias instancias con diferentes valores de $q1$. Estos problemas están todos compuestos por 5 sub-CSPs, cada uno de los cuales tiene 15 variables con una talla de dominio de 10 ($< 5, 15, 10, 0.2, 0.01, 0.2, q1, q2 >$). Se puede observar que:

- Independientemente de $q1$ y $q2$, DTS mantiene un mejor comportamiento que FC.

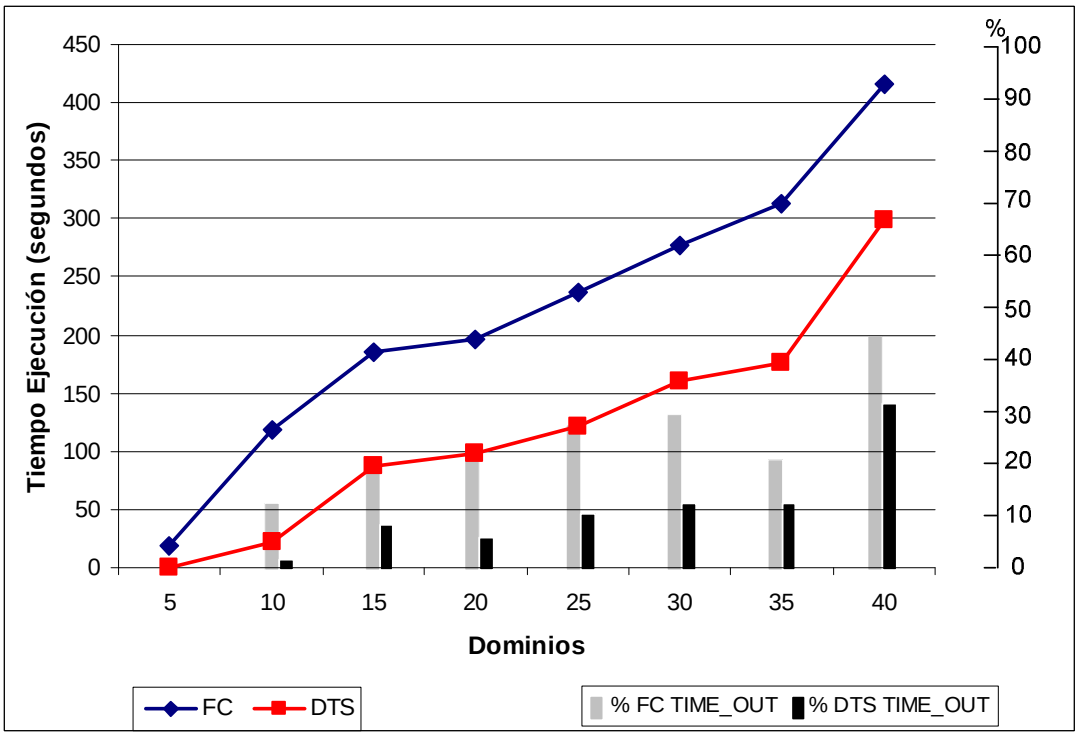


Figura 7.2: Tiempos de ejecución y porcentaje de soluciones que superan el $TIME\_OUT$ en problemas con diferente talla de dominio.

- A medida que incrementa el valor de $q1$, en general también aumentan la complejidad del problema y los tiempos de ejecución para la resolución del mismo. Sin embargo, a medida que la complejidad del problema aumenta, DTS tiene un mejor comportamiento en comparación con FC, ya que cuando los valores de $q1$ aumentan, los tiempos de ejecución de DTS son mucho mejores que los de FC.

- La complejidad del problema también depende de $q2$. En relación a $q2$, los problemas más complejos son aquellos generados, con valores medios de $q2$ ($q2 = 0.3$ y $q2 = 0.5$). Además, para valores altos de $q1$, los problemas resultaron también ser bastante complejos con valores bajos de $q2$ ($q2 = 0.1$). En todos estos casos (los de mayor complejidad) es cuando DTS tiene su mejor comportamiento.

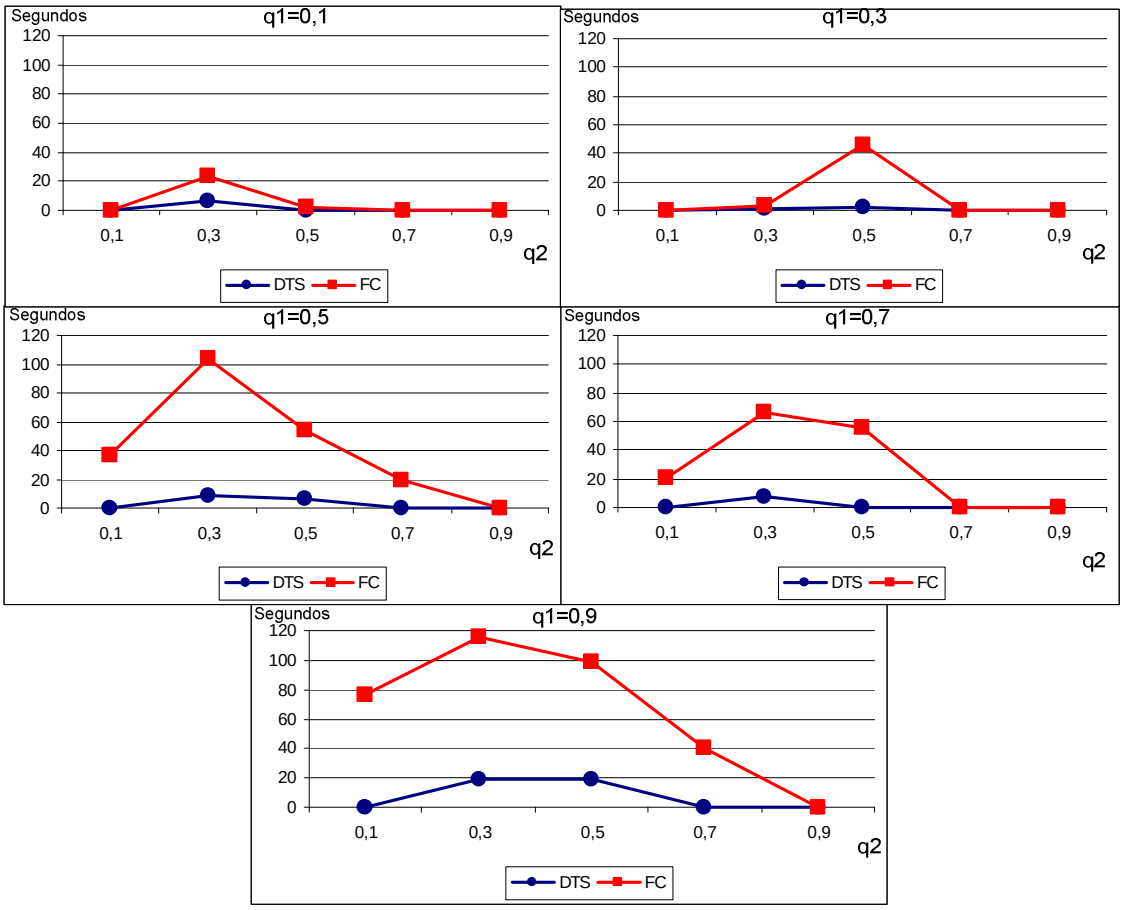


Figura 7.3: Tiempos de ejecución con diferentes q1 y q2.

7.2. Benchmarks de problemas de planificación ferroviaria

En esta sección llevamos a cabo una evaluación empírica sobre problemas de planificación ferroviaria modelados como CSPs. Comparamos las diferentes técnicas de particionamiento de CSPs, resueltas con el algoritmo distribuido DTS, y con conocidos algoritmos de resolución centralizada.

7.2.1. Introducción

La planificación ferroviaria es una tarea difícil y con grandes requerimientos de tiempo, particularmente en el caso de redes ferroviarias reales, donde el número y la complejidad de las restricciones crece drásticamente. Un horario ferroviario factible debe especificar el instante de llegada y de salida de cada tren a cada estación de su trayecto, de tal manera que la capacidad de la línea y todas las restricciones operativas sean tomadas en cuenta.

Tradicionalmente, los horarios ferroviarios son generados manualmente dibujando el trayecto de los trenes en un diagrama de espacio-tiempo llamado malla-ferroviaria. El horario de los trenes comienza en un instante de salida fijado por el operador y es manualmente ajustado de tal manera que se vayan cumpliendo las restricciones, trazando los trenes según su orden de prioridad. Llevar a cabo la planificación de todos los trenes es una tarea que puede tomar varios días; normalmente se encuentra un horario factible, pero no suele ser el más óptimo posible.

Un ejemplo de malla-ferroviaria se muestra en la Figura 7.4, donde varios cruces de trenes pueden ser observados. Una malla-ferroviaria contiene información relativa a la infraestructura ferroviaria (estaciones, vías, distancias entre estaciones, señales de control, etc.) y el horario de los trenes que usan la infraestructura (tiempo de llegada y salida de los trenes a las estaciones, frecuencias, paradas, cruces, etc.). Los nombres de las estaciones se muestran en el lado izquierdo de la Figura 7.4, y la línea vertical representa el número de vías entre estaciones (vía única o vía doble). La línea horizontal representa el tiempo.

La literatura de los años 60s, 70s y 80s relativa a la optimización ferroviaria es relativamente limitada. Comparada con la industria aeronáutica o el transporte por carretera, la optimización no fue generalmente tenida en cuenta en favor de la simulación o los métodos heurísticos. Sin embargo, Cordeau et al. [27] señalaron el aumento competitivo, la privatización, la desregularización, y el incremento en la velocidad de computación como razones importantes para usar técnicas de optimización en la industria ferroviaria. Una revisión de

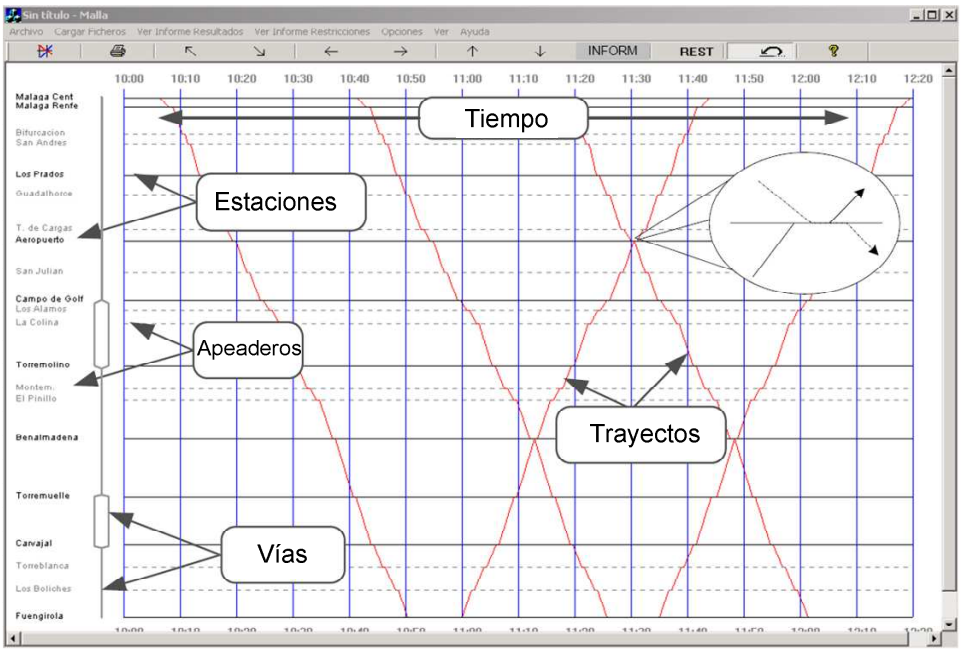


Figura 7.4: Ejemplo de malla-ferroviaria

métodos y modelos que han sido publicados indica que la mayoría de autores usa modelos que se basan en problemas de planificación de eventos periódicos (PESP) introducido por Serafini y Ukovich [101]. Los PESP consideran el problema de planificación como un conjunto de eventos que periódicamente vuelven a suceder bajo restricciones periódicas de tiempo. El modelo genera restricciones disyuntivas que podrían causar un crecimiento exponencial de la complejidad computacional del problema. En [93] se muestra un modelo para trenes periódicos que tiene en cuenta la topología de la infraestructura. Schrijver y Steenbeek [100] han desarrollado CADANS, un algoritmo basado en programación de restricciones que encuentra horarios factibles para un conjunto de restricciones PESP. El escenario considerado por esta herramienta es diferente del escenario que nosotros usamos, por lo que los resultados no son fácilmente comparables. El problema de planificación ferroviaria puede también ser modelado como un caso especial del problema de planificación job-shop (Silva de Oliveira [105], Walker et al. [113]), donde los tramos de

trayecto de los trenes son considerados tareas que son planificadas sobre las vías ferroviarias que se consideran como recursos.

Nuestro objetivo en esta evaluación es modelar el problema de planificación ferroviaria como un problema de satisfacción de restricciones ([58], [54], [18], [11], [3]) y resolverlo usando técnicas de programación de restricciones ([108], [92], [99], [66]). Sin embargo, debido al gran número de variables y restricciones que este problema genera, nosotros distribuimos el modelo en un conjunto de semi-independientes sub-problemas de tal manera que encontramos una solución más eficientemente [9].

Variables y Restricciones del problema de planificación ferroviaria

Las variables del problema de planificación ferroviaria representan los instantes de llegada y salida de los trenes a las estaciones. El dominio de las variables es el tiempo, con una granularidad en minutos. Hay tres grupos de reglas de planificación en nuestro problema: normas de tráfico, requerimientos de usuario y restricciones relativas a la infraestructura ferroviaria. Una malla-ferroviaria válida debe satisfacer todas estas reglas. Estas reglas de planificación pueden ser modeladas usando las siguientes restricciones, donde la variable $TA_{i,k}$ representa que el tren i llega a la estación k y la variables $TD_{i,k}$ representa que el tren i parte de la estación k:

1. Las **Normas de Tráfico** garantizan la correcta gestión de las operaciones de cruce y adelantamiento realizadas por los trenes. Las principales restricciones que se deben tener en cuenta son:

   - ***Restricción sobre el Tiempo de Recepción***. Existe un tiempo necesario para apartar un tren de la vía principal de manera que los cruces o adelantamientos se puedan realizar con seguridad (RecT).

$$(TA_{i,k} + RecT_i < TA_{j,k}) \vee (TA_{j,k} + RecT_j < TA_{i,k})$$

- ***Restricción sobre el Tiempo de Expedición***. Existe un tiempo necesario para colocar a un tren en la vía principal después de haber llevado a cabo la gestión de un cruce o adelantamiento (ExpT).

$$(TD_{i,k} + ExpT_i < TD_{j,k}) \vee (TD_{j,k} + expT_j < TD_{i,k})$$

- ***Restricción sobre cruce***: Cualquier par de trenes circulando en dirección opuesta no pueden usar simultáneamente un tramo de vía única.

$$(TD_{i,k} + T_{i,(k\_k+1)} < TD_{j,k+1}) \wedge (TD_{i,k} < TD_{j,k+1} + T_{j,(k+1\_k)})$$

$$\vee$$

$$(TD_{i,k} + T_{i,(k\_k+1)} > TD_{j,k+1}) \wedge (TD_{i,k} > (TD_{j,k+1} + T_{j,(k+1\_k)})$$

- ***Restricción sobre adelantamiento***: Dos trenes (i y s) circulando en el mismo sentido pero a diferentes velocidades solo pueden adelantarse en una estación.

$$(TD_{i,k} < TD_{s,k}) \wedge (TD_{i,k} + T_{i,(k\_k+1)} < TD_{s,k} + T_{s,(k\_k+1)})$$

$$\vee$$

$$(TD_{i,k} > TD_{j,k}) \wedge (TD_{i,k} + T_{i,(k\_k+1)} > (TD_{s,k} + T_{s,(k\_k+1)})$$

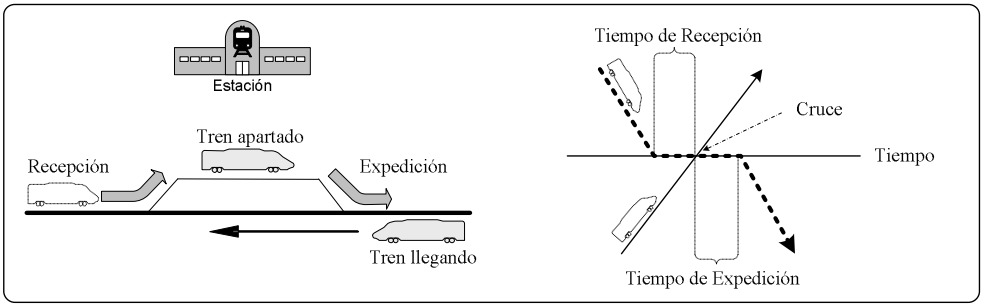


Figura 7.5: Restricciones sobre cruce y adelantamiento

2. **Requerimientos de usuario**: Las principales restricciones debidas a los requerimientos de usuario son:

   - ***El tipo y número de trenes*** circulando en cada sentido que deben ser planificados.
   - ***El trayecto de los trenes***: Las dependencias usadas y las ***paradas comerciales*** realizadas.

   $$TD_{i,k} = TA_{i,k} + StopTime_{i,k}$$

   - ***La Frecuencia de planificación***. La salida de los trenes debe satisfacer unos requerimientos de frecuencia en cada dirección. Esta frecuencia puede ser un tiempo exacto (1) o un intervalo de tiempo ($Freq \pm \delta$) (2). La frecuencia es una restricción muy dura y para algunos tipos de trenes no suele ser requerida.

     (1) $TD_{i+1,k} = TD_{i,k} + Freq$

     (2) $(TD_{i,k} + Freq - \delta) \leq TD_{i+1,k} \leq (TD_{i,k} + Freq + \delta)$
   - ***Intervalo de salida*** para la salida del primer tren circulando en ambos sentidos.

   $$StartTime_i < TD_{i,1} < EndTime_i$$

3. **La infraestructura ferroviaria y los tipos de trenes que la usan** dan lugar a un nuevo conjunto de restricciones que deben ser tomadas en cuenta. Algunas de ellas son:

   - ***Número de vías en las estaciones*** (para realizar operaciones técnicas y comerciales) y el ***número de vías entre dos estaciones*** (vía única o vía doble).
   - ***Restricciones de tiempo***, entre cada par de estaciones contiguas ($T_{i,(k\_k+1)}$).

$$TA_{i,k+1} - TD_{i,k} = T_{i,(k\_k+1)}$$

En las siguientes secciones evaluaremos las tres técnicas de particionamiento propuestas en esta tesis (particionamiento de grafos, identificación de árboles e identificación de entidades) junto con el algoritmo distribuido DTS presentado en el capítulo 5.

La evaluación empírica la llevamos a cabo sobre una infraestructura ferroviaria que une dos importantes ciudades españolas (La Coruña y Vigo). El tramo ferroviario entre estas dos ciudades está actualmente dividido en 40 estaciones. En nuestra evaluación cada conjunto de instancias aleatorias está definido por la 3-tupla $< n, s, f >$, donde n es el número de trenes periódicos circulando en cada sentido, s representa el número de estaciones y f es la frecuencia requerida entre la salida de trenes consecutivos. Los problemas fueron generados aleatoriamente modificando estos tres parámetros.

Normalmente, en este tipo de problemas, incrementar el número de trenes implica un CSP con un mayor número de variables; incrementar el número de estaciones implica un CSP con un mayor número de variables y con una talla del dominio de las variables también mayor; y decrementar la frecuencia implica incrementar la complejidad del problema porque el número de conflictos entre los trenes es mayor.

7.2.2. Particionamiento de grafos

En esta sección llevamos a cabo una evaluación entre nuestro algoritmo distribuido DTS y Forward Checking (FC), realizando un particionamiento del problema de scheduling basado en el particionamiento de grafos y usando para ello la herramienta METIS. La Figura 7.6 muestra una distribución de una malla-ferroviaria llevada a cabo por METIS. El agente rojo está encargado de asignar valores a las variables de los trenes 0 y 1 al comienzo y final de sus respectivos trayectos. El agente verde estudia dos trenes en partes disjuntas

del trayecto de los trenes. Observando visualmente el resultado del particionamiento llevado a cabo por METIS, da la impresión de que podría no ser el más apropiado, ya que agrupa en un mismo agente variables de tramos y trenes dispares. Esto es debido a que METIS es un software general que no tiene en cuenta información adicional sobre el problema para guiar el proceso de particionamiento.

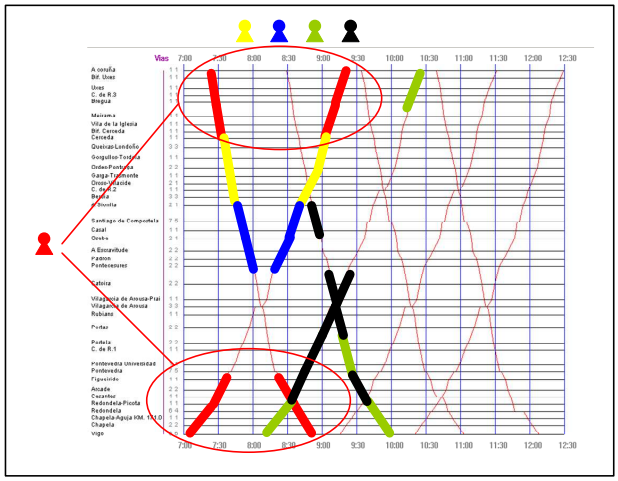


Figura 7.6: Distribución del problema de planificación ferroviaria basado en el particionamiento de grafos.

Hemos visto en la sección previa que el software de particionamiento de grafos obtenía muy buenos resultados en problemas aleatorios. Sin embargo, en el problema de planificación ferroviaria, los resultados que se obtienen no son buenos. La Figura 7.7, muestra los resultados obtenidos con FC y con DTS para varias instancias de $< n, 20, 120 >$. Cuando el número de trenes es bajo, el comportamiento del algoritmo distribuido DTS es mejor que el del algoritmo centralizado FC. Sin embargo, con más de 8 trenes, el modelo distribuido fue incapaz de resolver el problema en mas de 1000000 segundos, por lo que la ejecución fue abortada. Estudiando las particiones generadas por METIS comprobamos que el trayecto de un tren es particionado entre varios sub-problemas, y cada sub-problema está compuesto por tramos de trenes circulando en sentidos opuestos. Cada una de las particiones resultantes son

fáciles de resolver, pero es muy difícil de coordinar los resultados obtenidos por los diferentes agentes.

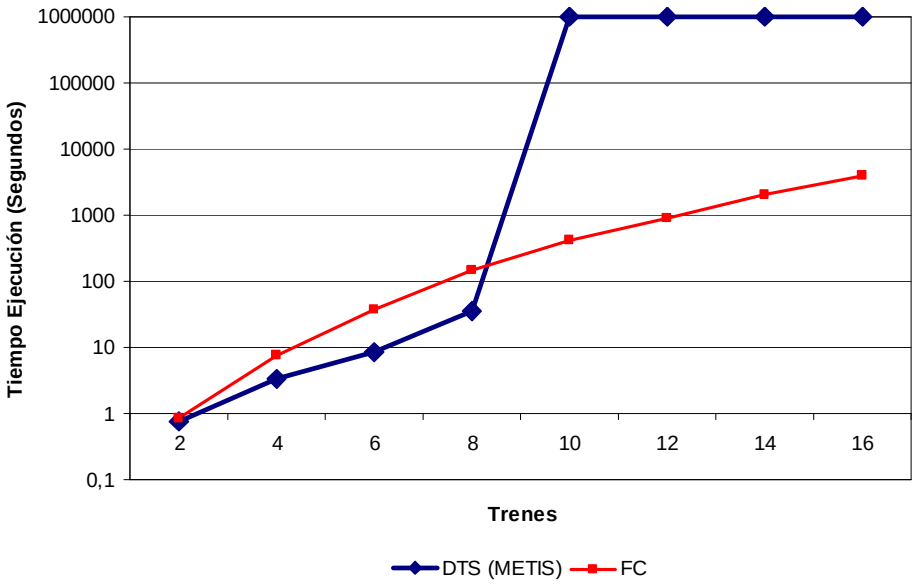


Figura 7.7: Comparación del tiempo de ejecución de FC y DTS usando el software de particionamiento METIS

7.2.3. Identificación de estructuras de árbol

En esta sección, llevamos a cabo una evaluación entre nuestro algoritmo distribuido DTS y un conocido algoritmo centralizado: FCPath. Este algoritmo realiza una consistencia de senda sobre las variables futuras cada vez que la variable actual es instanciada.

En esta evaluación comparamos los tiempos de ejecución y el número de chequeos de restricciones concurrentes (CCC) ([73]) del algoritmo DTS y FC-Path. DTS es ejecutado sobre dos diferentes tipos de particiones basadas en la identificación de árboles: la primera se basa en el algoritmo *ParticionarArboles* (Algoritmo 1), el cual es un método general de particionamiento de árboles (*árboles aleatorios*); el segundo método de particionamiento de árboles (*árboles trenes*) es dependiente del dominio y se ilustra en las Figuras 7.8 y 7.9.

La Figura 7.8 muestra el conjunto de variables de dos trenes circulando en direcciones opuestas entre las estaciones A y E. Tras estudiar este problema, detectamos una ventajosa partición del problema de planificación ferroviaria basada en árboles. En la Figura 7.8, los ejes entre dos variables representan las *restricciones de tiempo* ($TD_{i,k}$- $TA_{i,k+1}$) y las *restricciones de tiempo de parada* ($TA_{i,k}$- $TD_{i,k}$) respectivamente; estas restricciones crean el trayecto de los trenes y podrían representar información privada de los operadores ferroviarios. La Figura 7.9 muestra una clara partición del problema de planificación ferroviaria basada en árboles, donde cada sub-CSP tiene todas las variables de un solo tren, y sus respectivas *intra-restricciones* son las *restricciones de tiempo* y las *restricciones de tiempo de parada*, que son normalmente fijadas por los operadores ferroviarios. Las *inter-restricciones* son las reglas de tráfico que normalmente son vigiladas por los controladores de infraestructura.

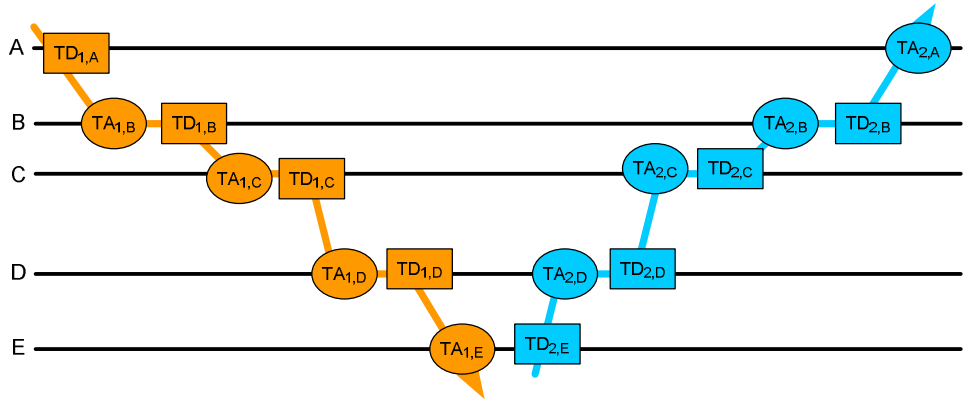


Figura 7.8: Variables de dos trenes circulando en direcciones opuestas.

Las Figuras 7.10 y 7.11 muestran el comportamiento de DTS y FCPath en varias instancias de n según la tupla $< n, 5, 60 >$. El número de trenes (n) fue incrementado de 1 a 20 trenes en cada sentido. Se debe tener en cuenta que las gráficas mantienen una escala logarítmica. Las Figuras 7.10 y 7.11 muestran que DTS tiene un mejor comportamiento que el algoritmo FCPath, tanto con *árboles aleatorios* como con *árboles trenes*, en todas las instancias. La Figura 7.10 muestra que DTS con *árboles trenes* siempre tiene menores tiempos de

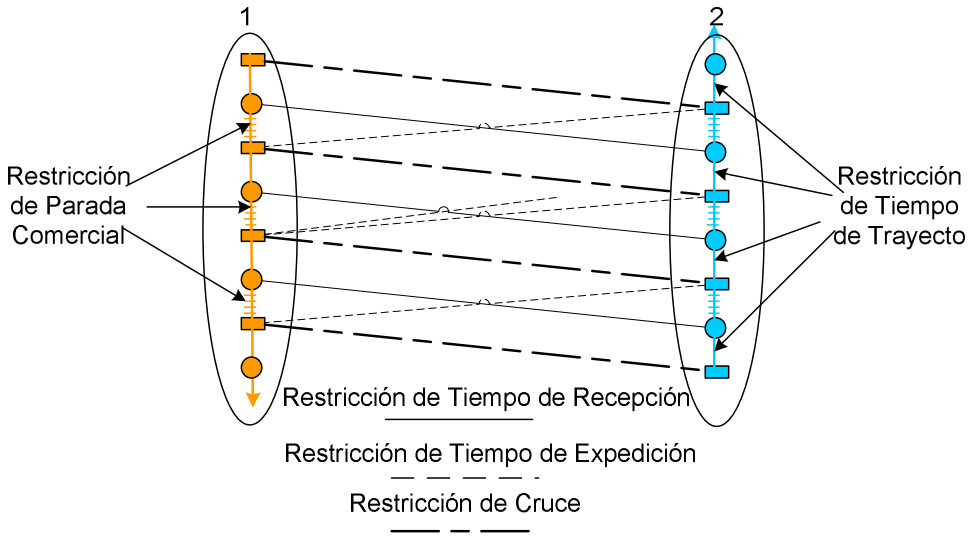


Figura 7.9: Partición de trenes basada en árboles.

ejecución que DTS con *árboles aleatorios*. Sin embargo, la Figura 7.11 muestra que DTS con *árboles aleatorios* algunas veces tiene menos CCC que DTS con *árboles trenes*. Las dos últimas afirmaciones parecen contradictorias, pero tiene una explicación viendo los resultados mostrados en la Figura 7.12: DTS con *árboles trenes* intercambia un menor número de mensajes que DTS con *árboles aleatorios*, por esto *árboles trenes* ahorra mucho tiempo de ejecución. Esto se debe a que *árboles trenes* implica una mejor coordinación entre los agentes.

La Figura 7.13 muestra los comportamiento de DTS y FCPath en varias instancias de s según la tupla $< 4, s, 60 >$, donde el número de estaciones (s) es incrementado de 5 a 20. Se puede observar que DTS con *árboles trenes* tiene un mejor comportamiento que DTS con *árboles aleatorios* y que FCPath. En la Figura 7.14, evaluamos el comportamiento de los algoritmos con diferentes frecuencias según la tupla $< 4, 10, f >$, donde la frecuencia es incrementada de 15 a 60 minutos. Se puede observar que debido al hecho de que todas las instancias mantienen el mismo número de variables y la misma talla del dominio, DTS con *árboles trenes* y FCPath mantienen comportamientos homogéneos. En general, ambos grafos corroboran el mejor comportamiento del algoritmo DTS, particularmente con la partición *árboles trenes*.

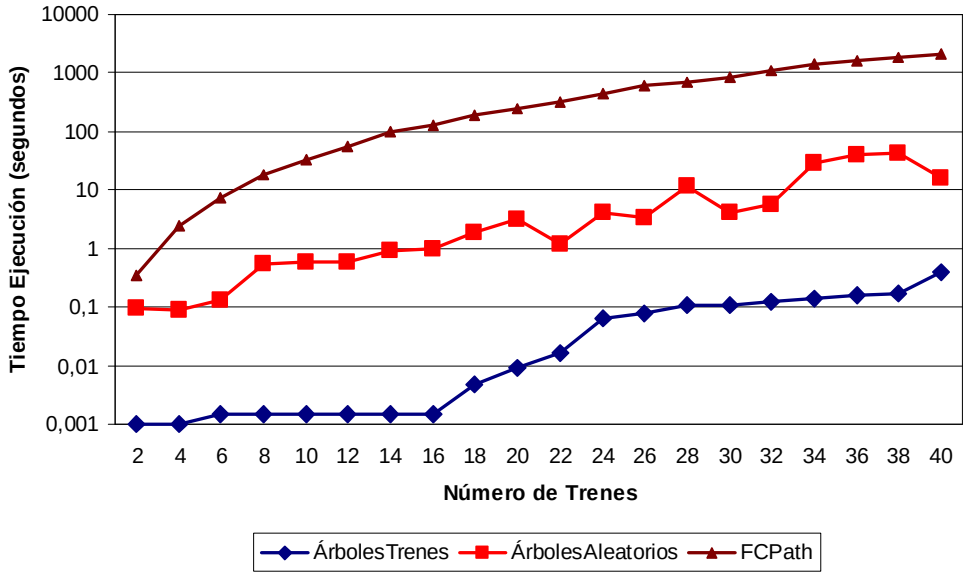


Figura 7.10: Tiempos de ejecución en los problemas $< n, 5, 60 >$.

7.2.4. Identificación de entidades

En esta sección, llevamos a cabo una evaluación entre nuestro algoritmo distribuido DTS y FC. En esta ocasión el problema es distribuido identificando entidades propias del problema. Este tipo de particionamiento requiere un estudio en profundidad del problema para extraer información adicional de la topología del problema de planificación ferroviaria y así obtener una mejor distribución del problema. Del estudio de este problema hemos deducido dos entidades principalmente:

- ***Trenes***. El problema puede ser considerado como un problema de scheduling donde los trenes son las tareas, de modo que la partición es llevada a cabo por medio de trenes. Cada agente está encargado de asignar valores a todas las variables de uno o varios trenes. Dependiendo del número de particiones deseadas, cada agente manejará uno o varios trenes. La Figura 7.15 muestra una malla-ferroviaria con 20 particiones donde cada agente maneja un único tren. Este modelo de particionamiento tiene dos importantes ventajas: Primero, este modelo nos permite mejorar la

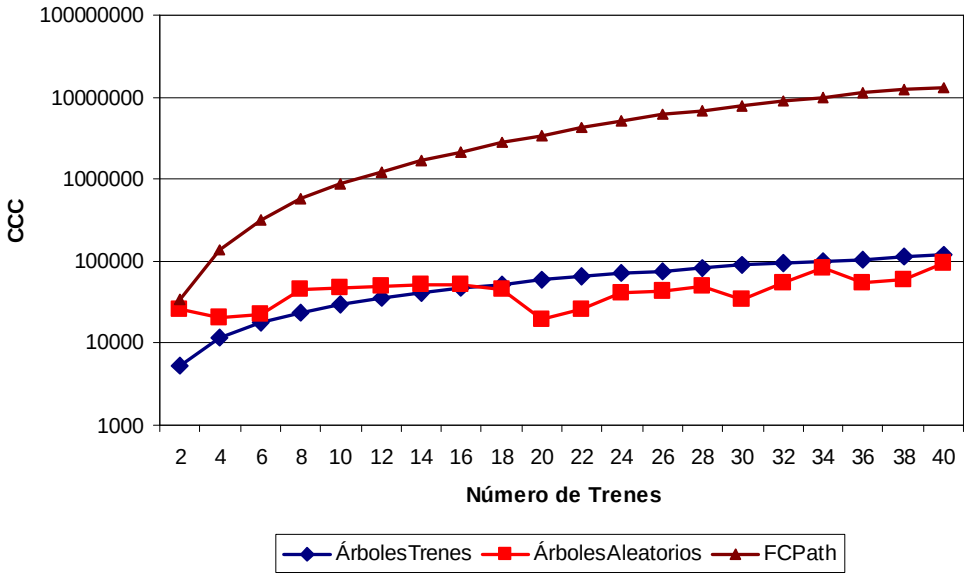


Figura 7.11: Chequeos de Restricciones Concurrentes (CCC) en los problemas $< n, 5, 60 >$.

privacidad. Actualmente, debido a la política de desregularización en las compañías europeas de ferrocarril, trenes de diferentes operadores ferroviarios circulan por la misma infraestructura ferroviaria. De este modo, este tipo de particionamiento nos da la posibilidad de distribuir el problema de manera que cada agente se encargue de los trenes de un operador. Así, diferentes operadores mantienen en privado sus datos estratégicos. La segunda ventaja es que este modelo nos permite manejar eficientemente las prioridades entre diferentes tipos de trenes (regionales, de alta velocidad, mercancías, etc.). De este modo, los agentes encargados de los trenes de mayor prioridad deberán realizar primero la asignación de los valores de sus variables para poder obtener un mejor tiempo de viaje.

- **Estaciones**. El problema puede ser considerado como un problema de scheduling donde las estaciones son los recursos del problema. Así, este tipo de particionamiento se basa en distribuir el problema por medio de estaciones contiguas. Debido a la desregularización de los operadores ferroviarios europeos, actualmente se planifican trenes con largos trayectos

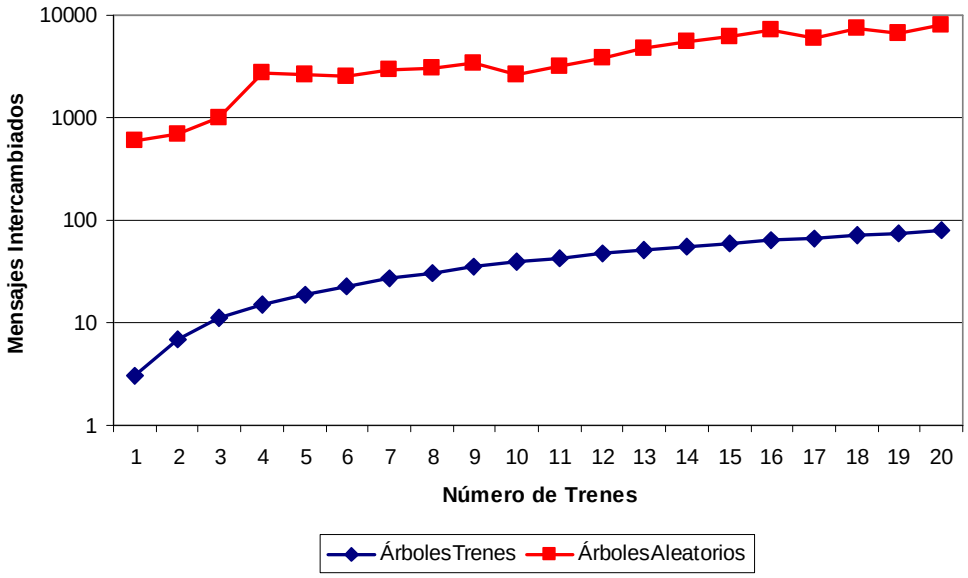


Figura 7.12: Número de mensajes intercambiados en problemas $< n, 5, 60 >$.

que involucran un gran número de estaciones las cuales pueden pertenecer a países diferentes con distintas políticas ferroviarias. Por lo tanto, una lógica partición de la red ferroviaria sería por regiones (estaciones contiguas). Para realizar este tipo de partición es importante analizar la infraestructura ferroviaria y detectar las regiones más congestionadas (cuellos de botella). Para equilibrar el problema no siempre lo importante es que todos los agentes manejen el mismo número de estaciones, sino que un agente puede manejar muchas estaciones si se encuentran en una zona de poco tráfico, mientras que otro agente maneja muy pocas estaciones si estas se encuentra en una zona que resulta ser un cuello de botella. Además los agentes encargados de gestionar estaciones muy congestionadas deberían tener preferencia para asignar los valores de sus variables debido a que sus restricciones limitan mucho sus dominios. Con este tipo de particionamiento, la malla-ferroviaria entre dos ciudades se divide en varias mallas-ferroviarias mas pequeñas. La Figura 7.16 muestra una malla-ferroviaria que ha sido particionada en bloques de estaciones contiguas de manera que cada bloque sea manejado por un

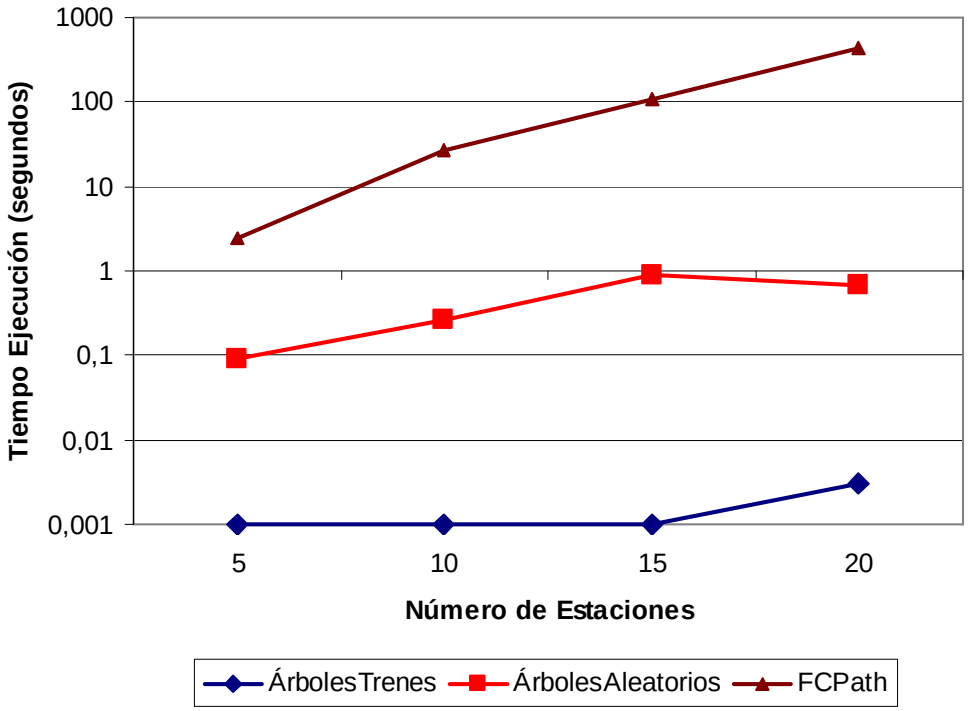


Figura 7.13: Tiempos de ejecución en problemas $< 4, s, 60 >$.

agente.

Las Figuras 7.17 y 7.18 muestran el comportamiento del algoritmo DTS con la partición basada en la entidad tren donde cada agente maneja un único tren. En ambas gráficas, se puede observar que el tiempo de ejecución incrementa cuando se aumenta el número de trenes (Figura 7.17) y cuando se aumenta el número de estaciones (Figura 7.18). De cualquier modo, en ambos casos, el modelo distribuido (DTS) mantiene un mejor comportamiento que el modelo centralizado.

La partición basada en la entidad tren tiene un comportamiento algo mejor, aunque muy similar a la partición basada en la entidad estación (ver Figura 7.19), principalmente cuando el número de trenes aumenta. Sin embargo, la partición basada en la entidad estación mantiene un comportamiento más uniforme.

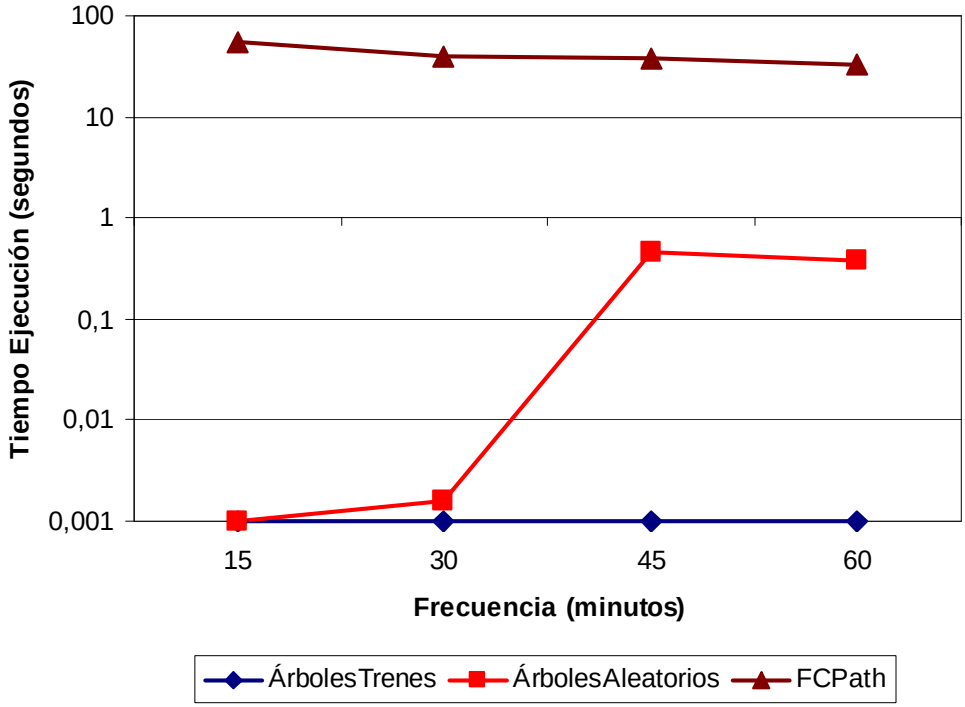


Figura 7.14: Tiempos de ejecución en problemas $< 4, 10, f >$.

7.3. Conclusiones

En este apartado hemos llevado a cabo una evaluación empírica de las diferentes técnicas de particionamiento de CSPs propuestas en el Capítulo 4. La evaluación ha consistido en una comparación de los tiempos de ejecución, el número de chequeos de restricciones y el intercambio de mensajes llevado a cabo en la resolución de los problemas mediante técnicas distribuidas y centralizadas. El algoritmo distribuido utilizado para resolver los problemas particionados ha sido el algoritmo DTS presentado en el Capítulo 5. Para resolver los problemas de modo centralizado se han usado diferentes versiones del algoritmo Forward Checking.

Tras analizar los resultados obtenidos en esta evaluación con problemas generados aleatoriamente se deduce que en general es una buena opción particionar los CSPs para resolverlos mediante técnicas de CSPs distribuidos, sobretodo cuando se trabaja con problemas de gran complejidad debida, bien

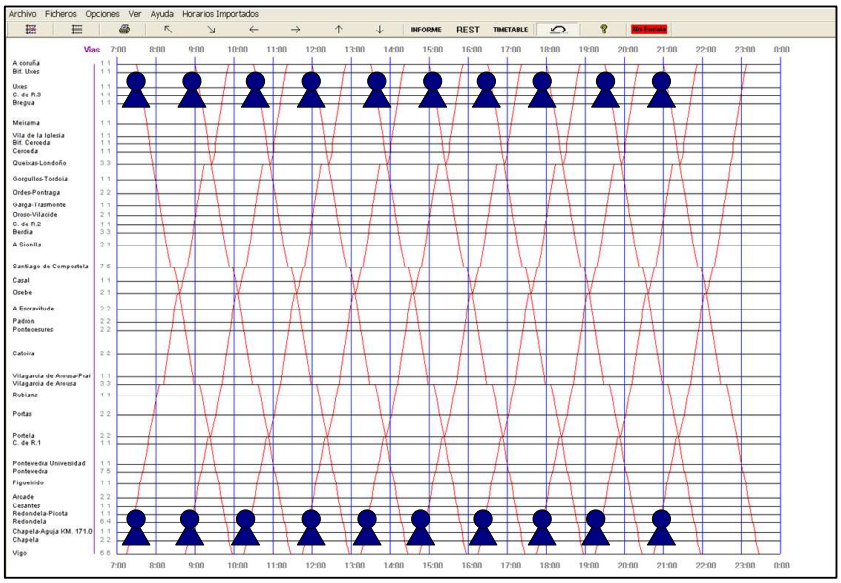


Figura 7.15: Distribución del problema de planificación ferroviaria mediante la entidad tren.

al gran número de variables o bien, a la complejidad de las restricciones. En estos casos tanto las técnicas de particionamiento de grafos como las de identificación de árboles resultan igualmente útiles, aunque los gráficos muestran que los mejores resultados se obtienen con las técnicas de particionamiento de grafos.

Los resultados obtenidos de la evaluación de benchmarks de problemas de planificación ferroviaria nos muestran que también es posible aplicar técnicas distribuidas para resolver problemas reales de naturaleza centralizada. Sin embargo, en estos casos, para poder resolver el problema de manera eficiente se requiere de un análisis previo del problema para ser capaces de identificar entidades que nos permitan particionar el problema de una manera adecuada.

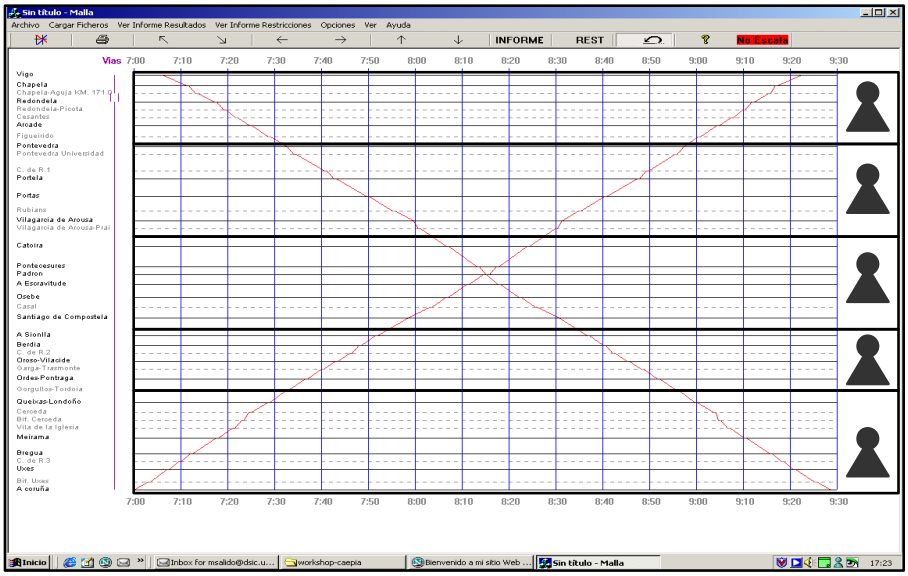


Figura 7.16: Distribución del problema de planificación ferroviaria mediante la entidad estación.

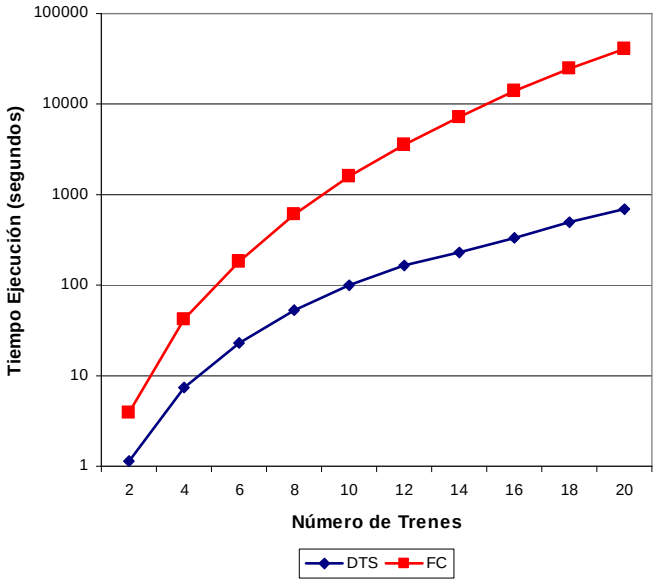


Figura 7.17: Tiempo de ejecución cuando aumenta el número de trenes.

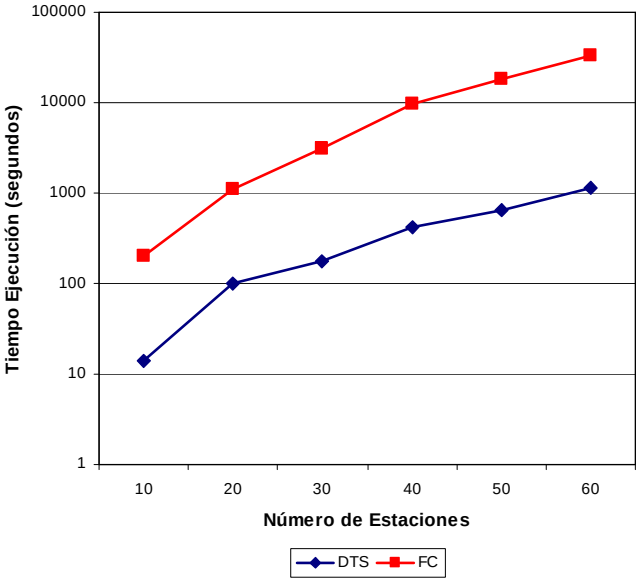


Figura 7.18: Tiempo de ejecución cuando aumenta el número de estaciones.

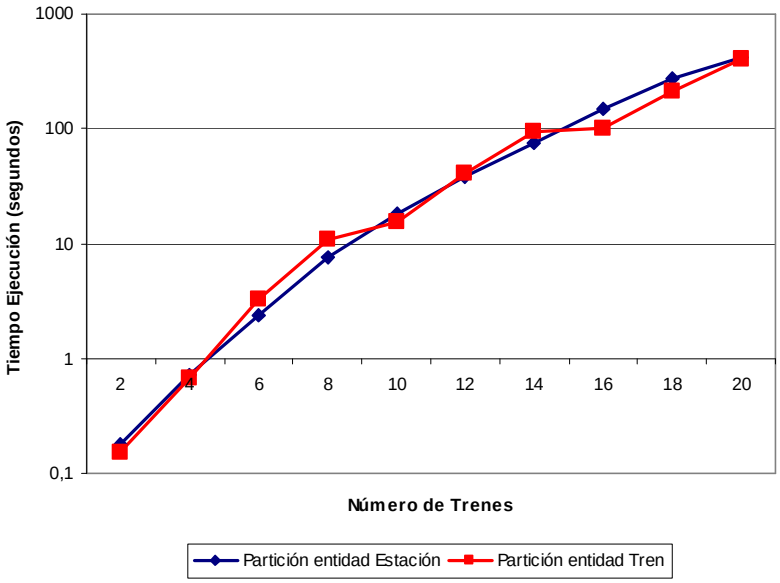


Figura 7.19: Comparación de las particiones basadas en las entidades tren y estación respectivamente .

Capítulo 8

Conclusiones y Trabajos Futuros

8.1. Contribuciones destacadas

Esta tesis doctoral consta principalmente de tres partes claramente diferenciadas:

- Propuestas de particionamiento de CSPs centralizados para su posterior resolución mediante técnicas de CSPs distribuidos. Contenido en el Capítulo 4.
- Diseño e implementación de algoritmos y técnicas distribuidas multivariable. Contenido en los Capítulos 5 y 6.
- Finalmente en el Capítulo 7 se han evaluado y contrastado empíricamente las técnicas anteriores sobre diversos casos y escenarios de prueba.

En una primera parte (Capítulo 4) se proponen las técnicas de particionamiento de grafos como una nueva técnica para resolver CSPs centralizados mediante técnicas de CSPs distribuidos. Esta técnica toma ventaja del criterio de mínima conectividad entre los sub-CSPs resultantes, con el objetivo de minimizar el paso de mensajes entre los agentes encargados de la resolución de los sub-CSPs. Las evaluaciones realizadas muestran que se pueden obtener buenos resultados usando estas técnicas con problemas de gran tamaño. La

siguiente propuesta de particionamiento se basa en la identificación de subestructuras que por sus características proporcionan a los agentes la habilidad de encontrar *soluciones_parciales* a los sub-CSPs con un bajo coste computacional, específicamente se tratan de estructuras con forma de árbol. De los resultados de la evaluación de este tipo de particionamiento vislumbramos que para ciertos tipos de problemas, como pueden ser los problemas de planificación ferroviaria, la técnica funciona muy bien, sobretodo cuando el problema tiene una gran complejidad y las estructuras de árbol seleccionadas son dependientes del problema (no aleatorias). Esto nos condujo a proponer como tercera técnica de particionamiento la identificación de entidades propias del problema como base para dividir el problema. Concretamente, estudiamos el problema de planificación ferroviaria, del cual deducimos que hay dos entidades representativas del mismo: trenes y estaciones. Los particionamientos del problema de planificación ferroviaria basados en estas dos entidades son los que mejores resultados obtuvieron, en cuanto a eficiencia, debido a que reducen en gran medida la comunicación entre los agentes, ya que la exactitud del particionamiento facilita la mejor coordinación entre los agentes.

Del particionamiento de grandes CSPs resultan sub-CSPs más pequeños pero los cuales todavía tienen un gran número de variables. La bibliografía revisada relativa a las técnicas de resolución de CSPs Distribuidos nos indicaba que la mayoría del esfuerzo de investigación realizado en este campo asumía que cada sub-CSP constaba únicamente de una única variable, y la adaptación de estas técnicas a sub-CSPs multivariable se hace inviable para grandes sub-CSPs debido bien a sus costes de cómputo, o bien a su requerimientos espaciales. Esto motivó el desarrollo de nuestra propuesta de algoritmo distribuido multivariable: DTS. Este algoritmo es capaz de manejar sub-CSPs con muchas variables de una manera muy eficiente y a la vez realizar una buena coordinación entre los agentes gracias a su organización en forma de *Estructura DFS-Tree CSP*. El algoritmo DTS es complementado con dos algoritmos centralizados de resolución de CSPs: FCA y TSA. Estos algoritmos facilitan

la búsqueda *intra-agente* de *soluciones_parciales* haciendo uso de la información recibida durante el proceso de comunicación realizado por el algoritmo DTS. FCA es un algoritmo de uso general y TSA es un algoritmo que solo puede ser usado para resolver sub-problemas con estructura de árbol. Además proponemos varias versiones heurísticas no completas de estos algoritmos que aceleran el proceso de búsqueda *intra-agente* sin necesidad de perder muchas soluciones.

Por lo tanto, podemos resumir el trabajo realizado en los siguientes puntos:

- Propuesta de particionamiento de CSPs mediante el uso de software de particionamiento de grafos tales como METIS.
- Creación de un algoritmo genérico para la identificación de estructuras de árbol en un CSP que conduzcan a un particionamiento del mismo.
- Propuesta de particionamiento de CSPs mediante la identificación de entidades propias del problema.
- Algoritmo de estructuración de un conjunto de sub-CSPs en una *Estructura DFS-Tree CSP*.
- Desarrollo e implementación de un nuevo algoritmo distribuido multivariable: DTS.
- Desarrollo e implementación de dos nuevos algoritmos centralizados para llevar a cabo la búsqueda *intra-agente*: FCA y TSA.
- Desarrollo de técnicas heurísticas para acelerar el proceso de búsqueda *intra-agente*: FCAH y TSAH.
- Evaluación experimental y comparativa con otros algoritmos para la resolución de CSPs.

8.2. Líneas Futuras de Desarrollo

El trabajo desarrollado en esta tesis doctoral no es algo cerrado sino que deja muchos caminos abiertos por los que continuar nuevas líneas futuras de investigación. Principalmente destacamos cuatro futuras líneas de investigación:

- Una de ellas sería la mejora en el proceso de comunicación del algoritmo DTS. Permitiendo una ejecución más asíncrona de los agentes, de manera que la comunicación por la *Estructura DFS-Tree CSP* no solo se realice de padres a hijos, sino que un agente se pueda comunicar con todos sus descendientes, para que de este modo se pueda detectar en fases más tempranas las incompatibilidades entre las asignaciones.

- Otra línea de investigación que queda abierta y en la que hay muy poco trabajo realizado es la referente al manejo de grandes sub-CSPs. Nuevas técnicas y heurísticas para la interpretación, creación o almacenamientos de Nogood relativos a grandes sub-CSPs que permitan, sin grandes requerimientos de espacios, ser capaces de aprovechar al máximo la información recibida durante el proceso de comunicación.

- La tercera línea de investigación que permanece abierta es referente al particionamiento del problema. Muchos trabajos se están realizando para subdividir problemas buscando la mínima conectividad entre los subproblemas. Sin embargo hay muy pocos trabajos relativos a la búsqueda de subproblemas con subestructuras eficientes.

- Finalmente, otra de las posibilidades abiertas sería la aplicación de las técnicas descritas en un entorno real distribuido para la gestión, por ejemplo, de la planificación ferroviaria interregional.

8.3. Publicaciones Relacionadas con la Tesis

En esta sección presentamos las publicaciones que están relacionadas con el trabajo de tesis doctoral. Estas publicaciones las hemos clasificado en: artículos en revista, LNCS y capítulos de libro, congresos internacionales y congresos nacionales.

Artículos en Revistas, LNCS y Capítulos de Libro

- [10] Nogood-FC for Solving Partitionable Constraint Satisfaction Problems. M. Abril, M. A. Salido, F. Barber. Journal of Intelligent Manufacturing (Ed. Springer), to appear, 2008. (JCR: 0.598 (2006))

- [9] Distributed Search in Railway Scheduling Problems. M. Abril, M. A. Salido, F. Barber. Engineering Applications of Artificial Intelligence (Ed. Elsevier Science), to appear, 2008. (JCR: 0.866 (2006))

- [4] An Assessment of Railway Capacity. M. Abril, F. Barber, L. Ingolotti, M. A. Salido, P. Tormos, A. Lova. Transportation Research Part E-Logistics and Transportation Review, (Ed. Elsevier Science), to appear, 2008. (JCR: 0.643 (2006))

- [95] Domain Dependent Distributed Models for Railway Scheduling. M. A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, A. Lova. Knowledge-Based Systems, 20, pp: 186-194, (Ed. Elsevier Science), 2007. (JCR: 0.576 (2006))

- [7] DFS-Tree Based Heuristic Search. M. Abril, M.A. Salido, F. Barber. The Seventh Symposium on Abstraction, Reformulation and Abstraction (SARA2007), LNAI 4612, pp: 5-19, 2007.

- [8] Distributed Models for solving CSPs. M. Abril, M. A. Salido, F. Barber, INFOCOMP Journal of computer science, ISSN: 1807-4545, pp: 43-50, 2007.

- [69] Intelligent Train Scheduling on a High-Loaded Railway Network. A. Lova, P. Tormos, F. Barber, L. Ingolotti, Miguel A. Salido, M. Abril. ATMOS 2004: Algorithmic Methods and Models for Optimization of Railways. Lecture Notes in Computer Science, Vol: 4359, pp: 219-232, 2007.

- [56] A scheduling order-based method to solve timetabling problems. L. Ingolotti, F.Barber, P. Tormos, A. Lova, M.A. Salido M. Abril. Selected from the XI Conf. de la Asociación Española de IA (CAEPIA´05), LNCS. Vol: 4177 pp: 52-61, 2006.

- [58] New Heuristics to Solve the CSOP Railway Timetabling Problem, L. Ingolotti, A. Lova, F. Barber, P. Tormos, M.A. Salido, M. Abril. The 19th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE'06), LNCS. Vol: 4031, pp: 400-409, 2006.

- [94] Domain Dependent Distributed Models for Railway Scheduling. M.A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, A. Lova, Applications and Innovations in Intelligent Systems XIV, Best Refereed Application paper. ISBN: 1-84628-665-4. pp: 163-176, 2006.

- [106] Distributed Constraint Satisfaction Problems to Model Railway Scheduling Problems, P. Tormos, M. Abril, M.A. Salido, F. Barber, L. Ingolotti, A. Lova. Computer in Railways X: Computer System Design and Operation in the Railway and Other Transit Systems. ISSN: 1743-3509. ISBN: 1-84564-177-9. pp: 289-297, 2006.

- [20] A Decision Support Sytem for Railway Timetabling (MOM): the Spanish Case. F. Barber, P. Tormos, A. Lova, L. Ingolotti, M.A. Salido, M. Abril. Computers in Railways X: Computer System Design and Operation in the Railway and Other Transit Systems, WIT Press, pp: 465-474, 2006.

- [5] Distributed Constraints for Large-Scale Scheduling Problems. M. Abril, M. A. Salido, F. Barber. Eleventh International Conference on Principles and Practice of Constraint Programming. CP 2005. LNCS 3709. Springer-Verlag. 2005.

- [11] A Heuristic Technique for the Capacity Assessment of Periodic Trains; M. Abril, M. A. Salido, F. Barber, L. Ingolotti, A. Lova, P. Tormos. Frontiers in Artificial Intelligence and Applications, IOS Press, Vol. 131, pp: 339-346, 2005.

- [93] Topological Constraints in Periodic Train Scheduling. M.A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, A. Lova. Planning, Scheduling and Constraint Satisfaction: From Theory to Practice. Frontiers in Artificial Intelligence and Applications (IOS Press), Vol 117, pp: 11-20, ISBN: 1-58603-484-7, ISSN: 0922-6389, 2005.

- [96] A Topological Model Based on Railway Capacity to Manage Periodic Train Scheduling. M.A. Salido, M. Abril, F. Barber, P. Tormos, A. Lova, L. Ingolotti. Applications and Innovations in Intelligent Systems XII. (Springer Verlag), ISBN: 1-85233-908-X. pp: 107-120. 2004.

- [54] An Efficient Method to Schedule New Trains on a High-Loaded Railway Network. L. Ingolotti, F. Barber, P. Tormos, A. Lova, Miguel A. Salido, M. Abril. Advances in Artificial Intelligence, LNCS/LNAI 3315 (Springer Verlag), pp: 164-173, 2004.

- [18] An Interactive Train Scheduling Tool for Solving and Plotting Running Maps. F. Barber, Miguel A. Salido, M. Abril, L. Ingolotti, A. Lova, P. Tormos. Current Topic in Artificial Intelligence, LNCS/LNAI, Vol 3040, pp: 646-655 (Springer Verlag). 2004.

Congresos Internacionales

- [6] Application of Meta-Tree-Based Distributed Search to the Railway Scheduling Problem. M. Abril, M.A. Salido, F. Barber. COPLAS'07:

CP/ICAPS 2007 Joint Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems, pp: 6-13, 2007.

- [12] Distributed Models in Railway Industry, M. Abril, M.A. Salido, F. Barber, L. Ingolotti, A. Lova, P. Tormos, Workshop on Industrial Applications of Distributed Intelligent Systems (INADIS 2006), 2006.

- [57] Train Scheduling: A Real Case. L. Ingolotti, F. Barber, P. Tormos, A. Lova, M.A. Salido, M. Abril. Algorithmic Methods for Railway Optimization (L. Kroon, F. Geraets, D. Wagner and C. Zaroliagis Ed.) ISSN: 1862-4405, vol 04261, 2006.

- [97] Optimization in Railway Scheduling. M.A. Salido, M. Abril, F. Barber, P. Tormos, A. Lova, L. Ingolotti. 2nd International Conference on Informatics in Control, Automation and Robotics (ICINCO´05), ISBN: 972-8865-29-5 , pp: 188-195. 2005.

- [3] Applying Analytical and Empirical Methods to the Assessment of Railway Capacity. M. Abril, F. Barber, L. Ingolotti, M.A. Salido, A. Lova, P. Tormos, J. Estrada. 5th Workshop on Algorithmic Methods and Models for Optimization of Railways, in ALGO'05. In Proceeding of ATMOS'05.

- [110] A Genetic Approach to Train Scheduling on a High-Traffic Railway Line, P. Tormos, A. Lova, L. Ingolotti, F. Barber, M. Abril, M.A. Salido. In Proceeding of 5th Workshop on Algorithmic Methods and Models for Optimization of Railways. 2005.

- [16] MOM, A Decision Support System for Railway Scheduling. F. Barber, M. Abril, L. Ingolotti, A. Lova, M. A. Salido, P. Tormos. Int. Trienal Conf. In OR/MS (IFORS'05), 2005.

- [67] An Efficient Heuristic Technique To Schedule New Trains On A High Loaded Network. A. Lova, P. Tormos, F. Barber, L. Ingolotti, M. Abril,

M. A. Salido. In Proceeding of IFOR´05. Int. Trienal Conf. In OR/MS (IFORS'05), 2005.

- [108] Periodic Single Track Railway Scheduling: A Sequential Approach. P. Tormos, A. Lova, F. Barber, L. Ingolotti, M. Abril, M.A. Salido. Int. Trienal Conf. In OR/MS (IFORS'05), 2005.

- [59] A Decision Support System (DSS) for the Railway Scheduling Problem. L. Ingolotti, P. Tormos, A. Lova, F. Barber, M.A. Salido, M. Abril. First IFIP Conference on Artificial Intelligence Applications and Innovations (Kluwer Academic Publishers), ISBN: 1-4020-8150-2, pp: 465-474. 2004.

- [92] Applying Topological Constraint Optimization Techniques to Periodic Train Scheduling. M.A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, A. Lova. In Proceeding of ECAI 2004 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems, pp: 17-26. 2004.

Congresos Nacionales

- [68] Técnicas para la programación del Tráfico Ferroviario Heterogéneo, A. Lova, P. Tormos, Barber, L. Ingolotti, M. Abril, M.A. Salido. XXIX Congreso Nacional de Estadística e Investigación Operativa, ISBN: 84-689-8553-8, pp: 535-536, 2006.

- [109] MOM: Un Sistema Software para la Optimización del Tráfico Ferroviario, P. Tormos, A. Lova, Barber, L. Ingolotti, M. Abril, M.A. Salido. XXIX Congreso Nacional de Estadística e Investigación Operativa, ISBN: 84-689-8553-8 pp: 633-634, 2006.

- [55] A Scheduling Order-Based Method to Solve the Train Timetabling Problem. L. Ingolotti, F. Barber, P. Tormos, A. Lova, M. A. Salido, and M. Abril. XI Conference of the Spanish Association for Artificial Intelligence (CAEPIA), ISBN:84-96474-13-5, Vol I, pp: 343-352, 2005.

- [98] Técnicas Distribuidas para Problemas de Scheduling a Gran Escala. M.A. Salido, M. Abril, F. Barber, P. Tormos, A. Lova, Laura Ingolotti. Caepia 2005 Workshop sobre Planificación, scheduling y Razonamiento Temporal, pp: 51-59, 2005.

- [99] Técnicas de Inteligencia Artificial en Planificación Ferroviaria. M. A. Salido, F.Barber, M. Abril, L. Ingolotti, A. Lova, P. Tormos, J. Estrada. VI Session on Artificial Intelligence Technology Transfer (TTIA '2005), Ed. Thomson, ISBN:84-9732-435-811-18. 2005.

- [66] Planificación del tráfico ferroviario en una red altamente sobrecargada. A. Lova, P. Tormos, F. Barber, L. Ingolotti, M. Abril, Miguel A. Salido. XXVIII Congreso Nacional de Estadística e Investigación Operativa, 2004.

- [107] Un sistema de ayuda a la toma de decisiones (DSS) para el problema de secuenciación de trenes periódicos. P. Tormos, A. Lova, F. Barber, L. Ingolotti, M. Abril, Miguel A. Salido. XXVIII Congreso Nacional de Estadística e Investigación Operativa, 2004.

- [19] An Interactive Train Scheduling Tool for Solving and Plotting Running Maps. F. Barber, M.A. Salido, L. Ingolotti, M. Abril, A. Lova, P. Tormos. V Session of Technology Transfer on Artificial Intelligence (TTIA), (Jose Cuena Award, Best paper of TTIA), ISBN: 84-8373-564-4, Vol(2), pp: 379-438, San Sebastian, 2003.

Bibliografía

[1] A. Abecker, R. Alami, C. Baral, T. Bickmore, E. Durfee, T. Fong, M.H. Goker, N. Green, M. Liberman, C. Lebiere, J.H. Martin, G. Mentzas, D. Musliner, N.Ñicolov, I.Ñourbakhsh, F. Salvetti, D. Shapiro, D. Scherekenghost, A. Sheth, L. Stojanovic, V. SunSpiral, and R. Wray. AAAI 2006 spring symposium reports. *AI Magazine*, 2006.

[2] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. *IEEE, In Proceedings of 20th International Parallel and Distributed Processing Symposium*, 2006.

[3] M. Abril, F. Barber, L. Ingolotti, M.A. Salido, A. Lova, P. Tormos, and J. Estrada. Applying analytical and empirical methods to the assessment of railway capacity. *In Proceeding of 5th Workshop on Algorithmic Methods and Models for Optimization of Railways*, 2005.

[4] M. Abril, F. Barber, L. Ingolotti, M.A. Salido, P. Tormos, and A. Lova. An assessment of railway capacity. *Transportation Research Part E-Logistics and Transportation Review, to appear*, 2008.

[5] M. Abril, M.A. Salido, and F. Barber. Distributed constraints for large-scale scheduling problems. *In Proceeding of Eleventh International Conference on Principles and Practice of Constraint Programming*, page 837.

[6] M. Abril, M.A. Salido, and F. Barber. Application of Meta-Tree-based distributed search to the railway scheduling problem. *COPLAS'07: CP/ICAPS 2007 Joint Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, pages 6–13, 2007.

[7] M. Abril, M.A. Salido, and F. Barber. DFS-Tree based heuristic search. *In Proceeding of The Seventh Symposium on Abstraction, Reformulation and Abstraction*, pages 5–19, 2007.

[8] M. Abril, M.A. Salido, and F. Barber. Distributed models for solving CSPs. *INFOCOMP Journal of computer science*, pages 43–50, 2007.

[9] M. Abril, M.A. Salido, and F. Barber. Distributed search in railway scheduling problems. *Engineering Applications of Artificial Intelligence, to appear*, 2008.

[10] M. Abril, M.A. Salido, and F. Barber. Nogood-FC for solving partitionable constraint satisfaction problems. *Journal of Intelligent Manufacturing, to appear*, 2008.

[11] M. Abril, M.A. Salido, F. Barber, L. Ingolotti, A. Lova, and P. Tormos. A heuristic technique for the capacity assessment of periodic trains. *Frontiers in Artificial Intelligence and Applications*, 131:339–346, 2005.

[12] M. Abril, M.A. Salido, F. Barber, L. Ingolotti, A. Lova, and P. Tormos. Distributed models in railway industry. *Workshop on Industrial Applications of Distributed Intelligent Systems*, 2006.

[13] A. Armstrong and E. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. *In Proceedings of International Joint Conference on Artificial Intelligence, IJCAI'97*, pages 620–625, 1997.

[14] A. Babanov, J. Collins, and M. Gini. Scheduling tasks with precedence constraints to solicit desirable bid combinations. pages 345–352, 2003.

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

[16] F. Barber, M. Abril, L. Ingolotti, A. Lova, M.A. Salido, and P. Tormos. MOM, a decision support system for railway scheduling. *Int. Trienal Conf. In OR/MS (IFORS'05)*, 2005.

[17] F. Barber and M.A. Salido. Introducción a la programación de restricciones. *Inteligencia Artificial: Revista Iberoamericana de Inteligencia Artificial.*, 20.

[18] F. Barber, M.A. Salido, M. Abril, L. Ingolotti, A. Lova, and P. Tormos. An interactive train scheduling tool for solving and plotting running maps. *Current Topic in Artificial Intelligence*, 3040:646–655, 2004.

[19] F. Barber, M.A. Salido, L. Ingolotti, M. Abril, A. Lova, and P. Tormos. An interactive train scheduling tool for solving and plotting running maps. *V Session of Technology Transfer on Artificial Intelligence*, pages 379–438, 2003.

[20] F. Barber, P. Tormos, A. Lova, L. Ingolotti, M.A. Salido, and M. Abril. A decision support sytem for railway timetabling (MOM): the Spanish case. *Computers in Railways X: Computer System Design and Operation in the Railway and Other Transit Systems*, pages 235–244, 2006.

[21] R. Béjar, C. Domshlak, C. Fernández, C. Gomes, B. Krishnamachari, B. Selman, and M. Valls. Sensor networks and distributed CSP: communication, computation and complexity. *Artificial Intelligence*, 161(1-2):117–147, 2005.

[22] C. Bessiere, A. Maestre, and P. Messeguer. Distributed dynamic backtracking. *In Proceeding of IJCAI'01: Workshop on Distributed Constraint Reasoning*, pages 9–16, 2001.

[23] C. Bessière and Régin J.C. Refining the basic constraint propagation algorithm. *In Proceeding of IJCAI-2001*, pages 309–315, 2001.

[24] D.A. Burke and K.N. Brown. Applying interchangeability to complex local problems in distributed constraint reasoning. *In proceedings of Workshop on Distributed Constraint Reasoning, AAMAS*, pages 132–1464, 2006.

[25] M. Calisti and B. Faltings. Constraint satisfaction techniques for negotiating agents. *In Proc. AAMAS-02 Workshop on Probabilistic Approaches in Search*, 2002.

[26] M. Calisti and N.Ñeagu. Constraint satisfaction techniques and softwate agents. *In proceedings of Agents and Constraints Workshop, AIIA*, 2004.

[27] J. Cordeau, P. Toth, and D. Vigo. A survey of optimization models for train routing and scheduling. *Transportation Science*, 32:380–446, 1998.

[28] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

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

[30] R. Dechter. Constraint networks (survey). *Encyclopedia Artificial Intelligence*, pages 276–285, 1992.

[31] R. Dechter. *Constraint Processing*. Morgan Kaufman, 2003.

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

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

[34] E. del Acebo and E. Muntaner. Una plataforma para la simulación de sistemas multi-agente programados con Agent0. *Conf. de la Asociación Española para la Inteligencia Artificial*, 2001.

[35] 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.

[36] R. Ezzahir, C. Bessiere, M. Belaissaoui, and El-H. Bouyakhf. DisChoco: A platform for distributed constraint programming. *In Proceedings of IJCAI-2007 Eighth International Workshop on Distributed Constraint Reasoning (DCR'07)*, pages 16–27, 2007.

[37] M. Fabiunke. Parallel distributed constraint satisfaction. pages 1585–1591, 1999.

[38] S. Fitzpatrick and L. Meertens. An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs. *In Proc. 1st Symp. on Stochastic Algorithms: Foundations and Applications*, pages 49–64, 2001.

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

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

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

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

[43] 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.

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

[45] Y. Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools*, 11(2):167–188, 2002.

[46] Y. Hamadi, C. Bessiére, and J. Quinqueton. Backtracking in distributed constraint networks. *In Proceedings of European Conference of Artificial Intelligence (ECAI-98)*, pages 219–223, 1998.

[47] M. Hannebauer. A formalization of autonomous dynamic reconfiguration in distributed constraint satisfaction. *Fundamenta Informaticae*, 43(1-4):129–151, 2000.

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

[49] B. Hendrickson and R.W. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing*, 1995.

[50] K. Hirayama and M. Yokoo. Local search for distributed sat with complex local problems. *In Proceedings of AAMAS'02*, 53:1119–1206, 2002.

[51] K. Hirayama, M. Yokoo, and K.P. Sycara. The phase transition in distributed constraint satisfaction problems: First results. *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 515–519, 2000.

[52] T. Hogg and B.A. Hubermann. Controlling caos in distributed systemes. *IEEE Tramsactions on Systems, Man, and Cybernetics*, 21:1325–1332, 1991.

[53] L. Hunsberger and B.J. Grosz. A combinatorial auction for collaborative planning. *Proceedings of the Fourth International Conference on Multi-Agent Systems*, pages 151–158, 2000.

[54] L. Ingolotti, F. Barber, P. Tormos, A. Lova, M.A. Salido, and M. Abril. An efficient method to schedule new trains on a high-loaded. *Advances in Artificial Intelligence*, 3315:164–173, 2004.

[55] L. Ingolotti, F. Barber, P. Tormos, A. Lova, M.A. Salido, and M. Abril. A scheduling order-based method to solve the train timetabling problem. *Proceedings of XI Conference of the Spanish Association for Artificial Intelligence*, I:343–352, 2005.

[56] L. Ingolotti, F. Barber, P. Tormos, A. Lova, M.A. Salido, and M. Abril. A scheduling order-based method to solve the train timetabling problem.

Lecture Notes in Conputer Science. Selected from Proceedings of The XI Conf. de la Asociación Española de IA, pages 52–61, 2006.

[57] L. Ingolotti, F. Barber, P. Tormos, A. Lova, M.A. Salido, and M. Abril. Train scheduling: A real case. *Algorithmic Methods for Railway Optimization*, 2006.

[58] L. Ingolotti, A. Lova, F. Barber, P. Tormos, M.A. Salido, and M. Abril. New heuristics to solve the CSOP railway timetabling problem. *Proceedings of The 19th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 400–409, 2006.

[59] L. Ingolotti, P. Tormos, A. Lova, F. Barber, M.A. Salido, and M. Abril. A decision support system (DSS) for the railway scheduling problem. *First IFIP Conference on Artificial Intelligence Applications and Innovations*, pages 465–474, 2004.

[60] G. Karypis and V. Kumar. Using METIS and parMETIS. 1995.

[61] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, pages 71–95, 1998.

[62] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Journal on Scientific Computing*, 20:359–392, 1999.

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

[64] V. Kumar. Algorithms for constraint satisfaction problems: a survey. *Artificial Intelligence Magazine*, 1:32–44, 1992.

[65] J.S. Liu and K. Sycara. Collective problem solving through coordinated reaction. *Proceedings of the IEEE International Conference on Evolutionary Computation*, 2:575–578, 1994.

[66] A. Lova, P. Tormos, F. Barber, L. Ingolotti, M. Abril, and M.A. Salido. Planificación del tráfico ferroviario en una red altamente sobrecargada. *XXVIII Congreso Nacional de Estadística e Investigación Operativa*, 2004.

[67] A. Lova, P. Tormos, F. Barber, L. Ingolotti, M. Abril, and M.A. Salido. An efficient heuristic technique to schedule new trains on a high loaded network. *In Proceeding of IFOR´05. Int. Trienal Conf. In OR/MS*, 2005.

[68] A. Lova, P. Tormos, F. Barber, L. Ingolotti, M. Abril, and M.A. Salido. Técnicas para la programación del tráfico ferroviario heterogéneo. *In Proceeding of XXIX Congreso Nacional de Estadística e Investigación Operativa*, 2006.

[69] A. Lova, P. Tormos, F. Barber, L. Ingolotti, M.A. Salido, and M. Abril. Intelligent train scheduling on a high-loaded railway network. *Algorithmic Methods and Models for Optimization of Railways, LNCS*, 4359:219–232, 2007.

[70] Yokoo M. and Hirayama K. Distributed constraint satisfaction algorithm for complex local problems. *Proceedings of the 3rd International Conference on Multi Agent Systems*, 1998.

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

[72] A. Maestre and C. Bessiere. Improving asynchronous backtracking for dealing with complex local problems. *Proceedings of ECAI'04*, pages 206–210, 204.

[73] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraint processing algorithms. *In Proc. 4th Workshop on Distributed Constraint Reasoning*, 2002.

[74] A. Meisels and I. Razgon. Distributed forward-checking with dynamic ordering. *In Proceedings of DCR Workshop IJCAI-01*, 2001.

[75] A. Meisels and R. Zivan. Asynchronous forward-checking for DisCSPs. *Constraint*, 12:131–150, 2007.

[76] P. Meseguer. Constraint satisfaction problems: An overview. *AI Communications*, 2(1):3–117, 1992.

[77] P. Meseguer and M.A. Jimenez. Distributed forward checking. *In Proceedings of CP 2000 Workshop on Distributed Constraint Reasoning*, 2000.

[78] G.L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32:265–279, 1986.

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

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

[81] P. Morris. The breakout method for escaping from local minima. *In Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 40–45, 1993.

[82] I. Muñoz. Studies of dynamics of physical agent ecosystems. *PhD. Thesis, University of Girona*, 2002.

[83] T.Ñguyen and Y. Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.

[84] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. *In Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 266–271, 2005.

[85] A. Petcu and B. Faltings. ODPOP: An algorithm for open/distributed constraint optimization. *Proceedings of the National Conference on Artificial Intelligence*, pages 703–708, 2006.

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

[87] P. Prosser. MAC-CBJ: maintaining arc-consistency with conflict-directed backjumping. *Technical Report 95/177*, 1995.

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

[89] 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.

[90] A.L. Rosenberg and L.S. Heath. *Graph Separators, with Applications*. Kluwer Academic Publishers, 2002.

[91] 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.

[92] M.A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, and A. Lova. Applying topological constraint optimization techniques to periodic train scheduling. *In Proceeding of ECAI 2004 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, pages 17–26, 2004.

[93] M.A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, and A. Lova. Topological constraints in periodic train scheduling. *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice. Frontiers in Artificial Intelligence and Applications*, 117:11–20, 2005.

[94] M.A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, and A. Lova. Domain dependent distributed models for railway scheduling. *Applications and Innovations in Intelligent Systems XIV*, pages 163–176, 2006.

[95] M.A. Salido, M. Abril, F. Barber, L. Ingolotti, P. Tormos, and A. Lova. Domain dependent distributed models for railway scheduling. *Knowledge-Based Systems*, 20:186–194, 2007.

[96] M.A. Salido, M. Abril, F. Barber, P. Tormos, A. Lova, and L. Ingolotti. A topological model based on railway capacity to manage periodic train scheduling. *Applications and Innovations in Intelligent Systems*, XII:107–120, 2004.

[97] M.A. Salido, M. Abril, F. Barber, P. Tormos, A. Lova, and L. Ingolotti. Optimization in railway scheduling. *2nd International Conference on Informatics in Control, Automation and Robotics*, pages 188–195, 2005.

[98] M.A. Salido, M. Abril, F. Barber, P. Tormos, A. Lova, and L. Ingolotti. Técnicas distribuidas para problemas de scheduling a gran escala. *Workshop sobre Planificación, scheduling y Razonamiento Temporal, Caepia 2005*, pages 51–59, 2005.

[99] 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.

[100] A. Schrijver and A. Steenbeek. Timetable construction for railned. *Technical Report, CWI, Amsterdam, The Netherlands*, 1994.

[101] P. Serafini and W Ukovich. A mathematical model for periodic scheduling problems. *SIAM Journal on Discrete Mathematics*, pages 550–581, 1989.

[102] M.C. Silaghi and B. Faltings. Openess in asynchronous constraint satisfaction algorithms. *In Proceedings AAMAS-02 Workshop on Distributed Constraint Reasoning*, 2002.

[103] M.C. Silaghi and D. Mitra. Distributed constraint satisfaction and optimization with privacy enforcement. *3rd IC on Intelligence Agent Technology*, 2004.

[104] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. *In Proceedings AAAI 2000*, pages 917–922, 2000.

[105] E. Silva de Oliveira. Solving single-track railway scheduling problem using constraint programming. *Phd Thesis. Univ. of Leeds, School of Computing*, 2001.

[106] P. Tormos, M. Abril, M.A. Salido, F. Barber, L. Ingolotti, and A. Lova. Distributed constraint satisfaction problems to model railway scheduling problems. *Computer in Railways X: Computer System Design and Operation in the Railway and Other Transit Systems*, pages 289–297, 2006.

[107] P. Tormos, A. Lova, F. Barber, L. Ingolotti, M. Abril, and M.A. Salido. Un sistema de ayuda a la toma de decisiones (DSS) para el problema de secuenciación de trenes periódicos. *XXVIII Congreso Nacional de Estadística e Investigación Operativa*, 2004.

[108] P. Tormos, A. Lova, F. Barber, L. Ingolotti, M. Abril, and M.A. Salido. Periodic single track railway scheduling: A sequential approach. *In Proceeding of Int. Trienal Conf. In OR/MS (IFORS'05)*, 2005.

[109] P. Tormos, A. Lova, F. Barber, L. Ingolotti, M. Abril, and M.A. Salido. MOM: Un sistema software para la optimización del tráfico ferroviario. *XXIX Congreso Nacional de Estadística e Investigación Operativa*, pages 633–634, 2006.

[110] P. Tormos, A. Lova, L. Ingolotti, F. Barber, M. Abril, and M.A. Salido. A genetic approach to train scheduling on a high-traffic railway line. *In Proceeding of 5th Workshop on Algorithmic Methods and Models for Optimization of Railways*, 2005.

[111] E. Tsang. *Foundation of Constraint Satisfaction.* Academic Press, London and San Diego, 1993.

[112] P. van Beek and X. Chen. A constraint programming approach to planning. *In Proc. of the National Conference on Artificial Intelligence (AAAI-99)*, pages 585–590, 1999.

[113] C. Walker, J. Snowdon, and D. Ryan. Simultaneous disruption recovery of a train timetable and crew roster in real time. *Comput. Oper. Res*, pages 2077–2094, 2005.

[114] R. Wallace and E. Freuder. Constraint-based reasoning and privacy/efficiency tradeoffs in multi-agent problem solving. *Artificial Intelligence*, 161:209–227, 2005.

[115] M. Yokoo. Weak-commitment search for solving constraint satisfaction problems. *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 313–318, 1994.

[116] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. *In International Conference on Distributed Computing Systems*, pages 614–621, 1992.

[117] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: formalization and algorithms. *In IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[118] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. pages 401–408, 1996.

[119] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3:185–207, 2000.

[120] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: reaching agreement without revealing private information. *Artificial Intelligence*, 161(1-2):229–245, 2005.

[121] W. Zhang, G. Wang, and L. Wittenburg. Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance. *In Workshop on Probabilistic Approaches in Search AAAI-2002*, pages 53–59, 2002.

[122] W. Zhang and L. Wittenburg. Distributed breakout revisited. *Proceeding of the 18th AAAI*, pages 352–357, 2002.

[123] R. Zivan and A. Meisels. Parallel backtrack search on DisCSP. *In Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR2*, 2002.