Universidad Rey Juan Carlos

El problema de la minimización de la anchura de corte en ordenaciones lineales: resolución exacta y heurística

TESIS DOCTORAL

Eduardo García Pardo

2011

Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Departamento de Ciencias de la Computación

El problema de la minimización de la anchura de corte en ordenaciones lineales: resolución exacta y heurística

Tesis Doctoral

Directores:

Dr. D. Abraham Duarte Muñoz

Dr. D. Juan José Pantrigo Fernández

Doctorando:

D. Eduardo García Pardo

2011

El Dr. D. Abraham Duarte Muñoz, Profesor Titular de Universidad del Departamento de Ciencias de la Computación de la Universidad Rey Juan Carlos, y el Dr. D. Juan José Pantrigo Fernández, Profesor Contratado Doctor del Departamento de Ciencias de la Computación de la Universidad Rey Juan Carlos, directores de la Tesis Doctoral «*El problema de la minimización de la anchura de corte en ordenaciones lineales: resolución exacta y heurística*» realizada por el doctorando D. Eduardo García Pardo,

HACEN CONSTAR:

que esta Tesis Doctoral reúne los requisitos necesarios para su defensa y aprobación.

En Móstoles, a 11 de Abril de 2011,

Dr. D. Abraham Duarte Muñoz Dr. D. Juan José Pantrigo Fernández

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Listado de acrónimos

AMS del inglés, *Adaptive Multi-Start*

BB del inglés, *Branch and Bound*

BD Búsqueda Dispersa

BFS del inglés, *Breadth First Search*

BW del inglés, *BandWidth*

CL del inglés, *Candidate List*

CM del inglés, *Combination Method*

CMP del inglés, *Cutwidth Minimization Problem*

CMS del inglés, *Content Management System*

COP del inglés, *Combinatorial Optimization Problem*

CPU del inglés, *Central Processing Unit*

CV del inglés, *Critical Vertices*

CW del inglés, *CutWidth*

DFS del inglés, *Depth First Search*

EB del inglés, *Edge Bisection*

FPTAS del inglés *Fully Polynomial-Time Approximation Scheme*

GA del inglés, *Genetic Algorithm*

GPR del inglés, *GRASP + Path Relinking*

GRASP del inglés, *Greedy Randomized Adaptive Search Procedure*

HB Harwell-Boeing

HC del inglés, *Heuristic Concentration*

IP del inglés, *Integer Programming*

JCR del inglés, *Journal Citeseer Report*

LB del inglés, *Lower Bound*

LNCS del inglés, *Lecture Notes in Computer Science*

LGPL del inglés, *Lesser General Public License*

LP del inglés, *Linear Programming*

MA del inglés, *Memetic Algorithm*

MIP del inglés, *Mixed-Integer Programming*

MIQP del inglés, *Mixed-Integer Quadratic Programming*

NLP del inglés, *NonLinear Programming*

OP del inglés, *Optimization Problem*

PR del inglés, *Path Relinking*

PTAS del inglés *Polynomial-Time Approximation Scheme*

QP del inglés, *Quadratic Programming*

RCL del inglés, *Restricted Candidate List*

SA del inglés, *Simulated Annealing*

SAT del inglés, *Satisfiability*

SN del inglés, *Search Number*

SS del inglés, *Scatter Search*

TS del inglés, *Tabu Search*

TSP del inglés, *Traveling Salesman Problem*

UB del inglés, *Upper Bound*

URL del inglés, *Uniform Resource Locator*

VLSI del inglés, *Very Large Scale Integration*

VNS del inglés, *Variable Neighbourhood Search*

A mis padres

Agradecimientos

«El agradecimiento que sólo consiste en el deseo es cosa muerta, como es muerta la fe sin obras».

Miguel de Cervantes

En primer lugar debo dar las gracias a mis directores, Juanjo y Abraham. Gracias por darme la oportunidad de trabajar con vosotros; gracias por vuestras enseñanzas y consejos; gracias por vuestra ayuda, por vuestra confianza y por vuestro apoyo; gracias por considerar mis opiniones y, sobretodo, gracias por preocuparos por mí. Os agradezco también que me dierais la «chapa de investigador» aunque me la quitarais de forma forzosa a la semana siguiente (dichosos experimentos). Numerosas veces os he visto debatir con otros colegas cómo se debe dirigir a alguien y debo deciros que, aunque tal vez sea porque no he tenido otros, si tuviera que elegir de nuevo repetiría la experiencia.

En segundo lugar, me gustaría dar las gracias al profesor Rafael Martí, sin cuya ayuda todo habría sido mucho más difícil. Gracias por haber hecho también tuyo este proyecto y por enseñarme que en la precisión está la diferencia.

A Patxi y a Mica, por sus consejos, por sus sugerencias y por su ayuda. Gracias por incluirme en todos vuestros proyectos y por descubrirme mundos inagotables de conocimiento. Cada día aprendo más a vuestro lado. Gracias por vuestra amistad y, sobretodo, gracias por estar conmigo en los momentos malos.

A Soto, porque con ella todo esto comenzó, y porque sin ella no habría sido lo mismo. Porque has sido una persona muy especial e importante en este proceso y porque siempre serás mi tutora.

A Alfonso (por sus agudos comentarios), a Almudena (por su apoyo y comprensión), a Ángel y Belén (por sus consejos), a Antonio (por su empatía y genuina forma de ser), a José Vélez (por sus sugerencias y por su amistad), a Mayte (por ser la mejor compañera de despacho que uno puede desear), a Paco (por sus enseñanzas acerca de la evolución del docente), a Rafa (por ese otro modo de ver el mundo), a Raúl (por ser el espejo en el que conocí los estados de ánimo de «la Innombrable»), a Sergio (compañero de fatigas y última visita de la noche), y al resto de compañeros del DCC sin cuyo apoyo todo esto no habría sido posible.

Gracias también a mis amigos y más que amigos: a Raúl (el del cole); a Granada, Javi, Marcos y Ryan (los de Mérida); a Álvaro, Héctor, Judith, Hugo, Domingo, Javi, Chema y Jorge (los de Salamanca); a Bhavin (el de Londres); a Jaime, Rus, Rodri, Ana, Javi (Macías), Javi (Rodríguez), Víctor, Fran, Feliu, Laura, Miguel, Carlos y David (los de la uni); a Sergio, Alberto, Álvaro, Jaime, Jesús, Víctor, José, Pedro, Javi, Adri, JuanRa, Eduardo, Elena y Santi (los de Madrid).

Me gustaría destacar a Javi (el hermano que nunca tuve), a Álvaro (por su cercanía y confianza), a Héctor (por su inconmensurable afinidad), a Judith y Granada (las chicas) y a Alberto (por todas esas largas conversaciones de las que tanto he disfrutado) y, sobretodo, a Sergio, por ser tan especial, por aguantarme y por estar siempre a mi lado.

Por último, a toda mi familia, por su apoyo incondicional y, sobretodo, a los más importantes, a aquéllos por quienes soy quien soy, a mis padres. Gracias por dejarme soñar y gracias por ser como sois, porque vuestra entereza y tenacidad son contagiosas.

Gracias por todo.

Resumen

La optimización es una disciplina que trata de encontrar solución a problemas de la vida cotidiana. Una gran cantidad de problemas que tienen interés en áreas científicas y tecnológicas pueden ser enunciados como problemas de optimización. Un problema de optimización es aquél en el que, habiendo muchas posibles soluciones y alguna forma clara de comparación entre ellas, se pretende encontrar la mejor de todas, es decir, la solución óptima.

Los problemas de Optimización Combinatoria son un tipo de problemas de optimización cuyas soluciones están formadas por números enteros. Existen numerosos problemas de Optimización Combinatoria para los que no se conocen algoritmos capaces de resolverlos en tiempo polinómico. No obstante, dado el interés práctico de muchos de ellos, es necesario disponer de técnicas eficientes para abordarlos. Las técnicas existentes en la actualidad se podrían clasificar en exactas y aproximadas. Las técnicas exactas son capaces de encontrar la solución óptima, pero requieren un tiempo de cómputo elevado, por lo que son inviables cuando el tamaño del problema es grande. De entre las técnicas aproximadas destacan los algoritmos heurísticos, capaces de encontrar soluciones de alta calidad en un tiempo de cómputo razonable, pese a no poder certificar si la solución encontrada es óptima, ni cómo de lejos está de ella.

El problema de la minimización de la anchura de corte en ordenaciones lineales es un problema NP-Difícil de Optimización Combinatoria y consiste en encontrar un etiquetado de los vértices de un grafo, de modo que se minimice el máximo número de aristas que sobrepasan el espacio entre cada dos vértices consecutivos, al ordenar los vértices del grafo sobre una línea recta. Este problema tiene aplicaciones en diseño de circuitos, migración y fiabilidad de redes de telecomunicaciones, representación automática de grafos y en recuperación de información.

En esta Tesis Doctoral se proponen algoritmos exactos y heurísticos para la resolución del problema de la minimización de la anchura de corte en ordenaciones lineales. Los algoritmos exactos propuestos están basados en la técnica de Ramificación y Acotación, para la que se proponen distintas cotas inferiores y varios recorridos del árbol de exploración. Respecto a los algoritmos heurísticos, se proponen heurísticas constructivas, de mejora y de combinación de soluciones, que son embebidas dentro de un esquema híbrido GRASP con Búsqueda Dispersa. Tanto los algoritmos exactos, como los algoritmos heurísticos, mejoran los resultados obtenidos por los métodos existentes en el estado del arte, sobre los conjuntos de instancias evaluados.

Abstract

Optimization area is devoted to the search for solutions to real-life problems. A large number of problems with interest in Science and Technology can be stated as optimization problems. Generally speaking, an optimization problem is that one with a large number of feasible solutions, a clear way of comparison among them, and where the objective is to find the best possible solution, i. e. the optimum solution.

Combinatorial Optimization problems is a family of optimization problems where solutions are composed by integer numbers. For many Combinatorial Optimization problems, there are not polynomial time algorithms able to solve them. However due to their practical interest it is necessary to find efficient methods to tackle them. Nowadays, available techniques include exact and approximation methods. Exact methods are able to find the optimum solution, but they need too much time to obtain it, so they are impractical when the problem is large. As far as the approximation techniques are concerned, heuristic methods are probably the most remarkable ones. Heuristics can find high quality solutions in very little time but its quality it is not provable.

The Minimum Cut Linear Arrangement problem, also named the «Cutwidth problem», is a NP-Hard problem belonging to the Combinatorial Optimization family. The problem consist of finding a linear layout of the vertices of a graph, so the maximum number of cuts of a line separating any two consecutive vertices, is minimized. This problem has practical applications in circuit design, network migration, network reliability, automatic graph drawing and information retrieval.

In this Doctoral Thesis, exact and heuristic algorithms are proposed to tackle the resolution of the Minimum Cut Linear Arrangement problem. The proposed exact algorithms are based on the Branch and Bound technique and include different lower bounds for the problem. Heuristic algorithms are based on the GRASP and Scatter Search methodologies and include constructive, local search and combination methods. Both proposals outperform the results obtained by the state-of-the-art methods over the sets of instances evaluated.

Capítulo 1

Introducción

En el primer capítulo de este documento se presenta el contexto en el que está enmarcada esta Tesis Doctoral. De forma más específica, se describe el problema que se desea resolver, el «problema de la minimización de la anchura de corte en ordenaciones lineales» y se presenta la motivación, hipótesis de trabajo y objetivos. Además, se ofrece un esbozo de la propuesta algorítmica realizada para abordar el problema, que es detallada en sucesivos capítulos, así como del método de investigación empleado.

1.1. Problemas de optimización

La optimización es una disciplina que puede ser vista como piedra angular de otras áreas, tales como la Inteligencia Artificial, la Ciencia de la Computación o la Investigación Operativa [57]. La optimización trata de buscar soluciones factibles a problemas con aplicación en ingeniería, medicina, economía y otras muchas disciplinas científicas [30, 57].

Como área de investigación, la optimización ha crecido considerablemente en los últimos años debido, en gran parte, a la rápida evolución de los computadores. La constante tarea de tomar decisiones implica elegir entre varias alternativas en pro de tomar la mejor decisión posible.

Una gran cantidad de problemas que tienen interés en áreas científicas y tecnológicas pueden enunciarse como problemas de optimización. Se puede decir que un problema de optimización es un problema en el que hay varias (en general muchas) posibles soluciones y alguna forma clara de comparación entre ellas [51]. La calidad de cada una de dichas soluciones es evaluada por medio de la denominada función objetivo.

Desde el punto de vista matemático, un problema de optimización (OP, del inglés *Optimization Problem*) se puede definir como la minimización o maximización del valor de la función objetivo $f(x)$, ateniéndose a un conjunto de restricciones [142]. Más formalmente, un problema de minimización (sin pérdida de generalidad) se define como [184]:

$$\text{OP} = \begin{cases} \text{Minimizar} & f(x) \\ \text{sujeto a} & x \in F \end{cases}$$

donde F es el conjunto de soluciones que satisfacen todas las restricciones del problema. Cada $x \in F$ es denominada solución factible. De forma similar, una solución factible se denomina óptima, si el coste de la función objetivo de dicha solución es el menor, si el problema es de minimización, o el mayor, si el problema es de maximización, de entre todas las soluciones factibles.

Los problemas de Optimización Combinatoria (COP, del inglés *Combinatorial Optimization Problem*) son un tipo de problemas de optimización, cuyas soluciones están formadas por números enteros [8, 142]. Más formalmente [18], un COP $= (S, f)$ se podría definir como la maximización o minimización de $f(x)$ con S representando al espacio de soluciones factibles del problema. De manera resumida:

$$\text{COP} = \begin{cases} X = \{x_1, x_2, \ldots, x_n\} \\ D_1, \ldots, D_n \\ f : D_1 \times D_2 \times \ldots \times D_n \ \rightarrow \mathbb{R}^+ \\ S = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} | v_i \in D_i\} \end{cases}$$

donde cada variable x_i del conjunto X toma un valor v_i dentro del dominio D_i $(i = 1, \ldots, n)$ de manera que la solución $s \in S$ cumple con las restricciones del problema.

Los problemas de optimización también pueden ser clasificados según la «dificultad» que entraña su resolución para una computadora o, dicho de otro modo, de acuerdo con su complejidad computacional [62]. Para ello, se establecen las denominadas clases de complejidad, que agrupan familias de problemas para los que se observa una determinada propiedad relacionada con su complejidad. Para algunos problemas, existen algoritmos capaces de resolverlos en tiempo polinómico. Es decir, existe una función polinómica con exponente máximo p capaz de relacionar el tamaño de la entrada al algoritmo n con el tiempo máximo necesario (medido en iteraciones) para la resolución del problema. La complejidad de dicha función viene denotada por $O(n^p)$. Este tipo de problemas pertenecen a la denominada clase P. No obstante, existe una gran cantidad de problemas con interés práctico para los que no se conoce un algoritmo que los resuelva en tiempo polinómico, de manera exacta. De entre estos últimos, existen algunos para los que, en cambio, sí se puede determinar en tiempo polinómico si un valor corresponde a una solución del problema. Se dice que estos problemas pertenecen a la clase NP. Los problemas pertenecientes a la clase P también son problemas NP, ya que se puede determinar en tiempo polinómico si una solución dada es solución al problema. También, dentro de la clase NP, existen problemas para los que no se ha encontrado un algoritmo que los resuelva en tiempo polinómico, aunque no se ha podido demostrar que no exista. A estos problemas se los denomina NP-Completos. Los problemas NP-Completos presentan la peculiaridad de ser tan

difíciles como cualquier otro problema de la clase NP. Por lo tanto, si se demostrara que se puede resolver uno de ellos en tiempo polinómico, todos los problemas NP serían resolubles en tiempo polinómico [42]. Por último, existe una clase de problemas (NP-Difíciles) que, no perteneciendo necesariamente a la clase NP, son al menos tan difíciles como los problemas más difíciles de la clase NP [42, 62]. Para que un problema se considere NP-Difícil, éste debe poder ser reducido polinómicamente a algún problema de la clase NP-Completo.

1.2. Planteamiento y justificación del trabajo

Una vez definido el área de trabajo en el que se enmarca esta Tesis Doctoral, se pasará a describir más detalladamente el problema que se pretende abordar: el «*problema de la minimización de la anchura de corte en ordenaciones lineales*». Para ello, inicialmente se realiza una breve introducción a los problemas de ordenación lineal (entre los que se encuentra el problema en cuestión) y, a continuación, se define el problema en sí.

1.2.1. Problemas de ordenación lineal

Dentro de los problemas de Optimización Combinatoria existe un subconjunto de problemas que pueden ser formulados como problemas de ordenación lineal sobre grafos. Un grafo $G(V, E)$ es la representación de una relación [97]. Los grafos están compuestos por un conjunto de vértices (V) y un conjunto de aristas (E). Los vértices representan elementos de un conjunto y, las aristas, las relaciones entre dichos elementos. Cada arista es, por lo tanto, un segmento que conecta pares de vértices. En la Figura 1.1.a se muestra un ejemplo de representación de un grafo G con seis vértices y nueve aristas. Dado un grafo G, se puede obtener una ordenación lineal de sus vértices, donde cada vértice es etiquetado con un número entero de 1 a n, siendo n el número de vértices del grafo ($\mid V \mid = n$). En la Figura 1.1.b se muestra la representación de una posible ordenación lineal del mismo grafo de la Figura 1.1.a.

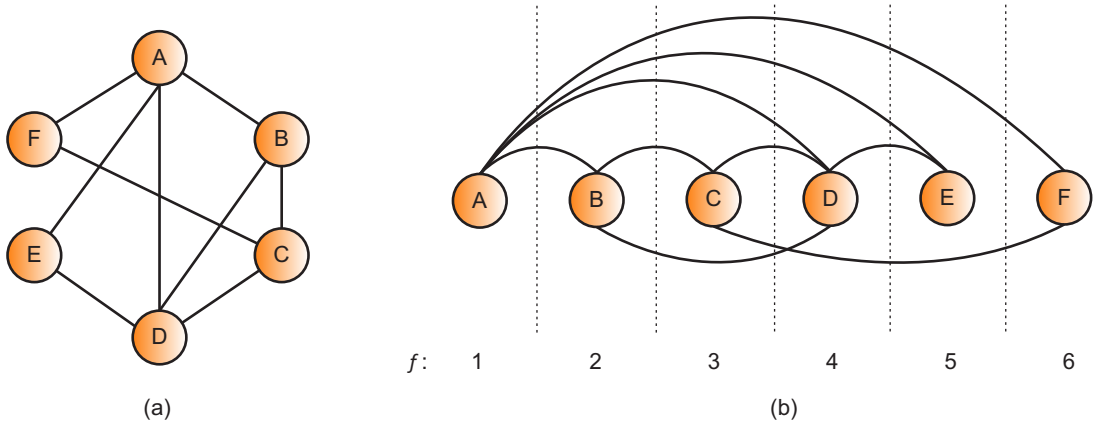


Figura 1.1: (a) Representación de un grafo $G(V, E)$. (b) Representación de una ordenación lineal de los vértices de G.

Los grafos son comúnmente utilizados para representar instancias de un problema. En optimización, se denomina instancia a un conjunto de datos numéricos que expresan un ejemplo, es decir, la asignación de un conjunto de valores concretos a cada uno de los parámetros de un problema.

Los problemas de ordenación lineal (*Linear Arrangement Problems*) son, por lo tanto, problemas consistentes en disponer los vértices de un grafo en una línea recta, estableciendo un orden para los mismos, de manera que una determinada función objetivo sea optimizada [46]. En otras palabras, dado un grafo G, se trata de encontrar una ordenación de los vértices de G, de modo que se maximice o minimice $f(x)$.

Algunos problemas de ordenación lineal, con aplicaciones prácticas en diversas disciplinas, pertenecen a la clase NP-Difícil. Entre los más conocidos, por su nombre en inglés, se encuentran los siguientes: *Antibandwidth, Bandwidth, Cutwidth, Minimum Linear Arrangement, Modified Cut, Profile, Sum Cut y Vertex Separation* [46].

1.2.2. El problema de la minimización de la anchura de corte en ordenaciones lineales

El problema de la minimización de la anchura de corte en ordenaciones lineales es un problema de Optimización Combinatoria; en concreto, es un problema de minimización. Se trata de un problema NP-Difícil [64] y, por lo tanto, su versión de decisión es NP-Completa [62]. Este problema es también conocido por su nombre en inglés como *Minimum Cut Linear Arrangement* o *Cutwidth Minimization Problem* (CMP) aunque también se pueden encontrar referencias a él como *Network Migration Scheduling* [3, 2] o *Board Permutation Problem* [40] (generalización para hipergrafos del problema). Por simplicidad, a lo largo de todo el documento, se hace referencia a él empleando el acrónimo CMP.

El CMP consiste en encontrar un etiquetado u ordenación de los vértices de un grafo, de modo que se minimice el máximo número de aristas que sobrepasan el espacio entre cada dos vértices consecutivos, al ordenar los vértices del grafo sobre una línea recta. Sea $G(V, E)$ un grafo no dirigido y no ponderado tal que V ($|V| = n$) es el conjunto de vértices y E ($|E| = m$) es el conjunto de aristas del mismo. Una ordenación lineal f de G asigna los enteros 1, 2,..., n a los vértices de G, de modo que cada vértice recibe una etiqueta diferente. Para dicha ordenación f, el valor del corte (denominado habitualmente *cutwidth*) de un vértice v, denotado por $CW_f(v)$, viene determinado por el número de aristas $(u, w) \in E$ tales que $f(u) \leq f(v) < f(w)$, es decir:

$$CW_f(v) = |\{(u, w) \in E : f(u) \leq f(v) < f(w)\}|$$

De este modo, el valor de la función objetivo para la ordenación f de G viene definido por la expresión:

$$CW_f(G) = \max_{v \in V} CW_f(v)$$

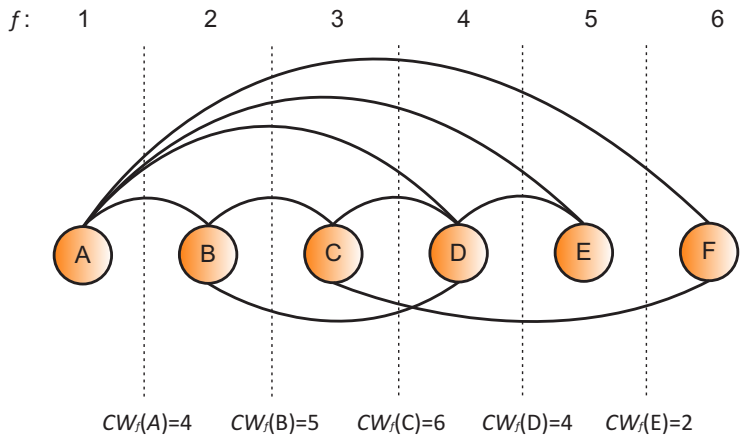


Figura 1.2: Representación gráfica de la ordenación lineal f de los vértices de G y del valor de $CW_f(v)$ de cada vértice.

En la Figura 1.2 se muestra una ordenación (f) de los vértices del grafo de la Figura 1.1.a y cada valor $CW_f(v) \, \forall \, v \in V$ en dicha ordenación, excepto para el vértice que ocupa la última posición cuyo *cutwidth* es igual a 0. Como se puede observar en la figura, el vértice A tiene un valor del *cutwidth* de 4, ya que existen cuatro aristas que cortan la línea recta imaginaria entre los vértices A y B. De igual modo, el vértice B tiene un *cutwidth* de 5 y así sucesivamente. El valor de la función objetivo del ejemplo viene determinado por el mayor de todos los cortes, es decir, el corte que genera el vértice que se encuentra en tercera posición, $CW_f(C) = 6$, luego el valor de la función objetivo para dicha ordenación sería $CW_f(G) = 6$.

En la Figura 1.3 se muestra otro ejemplo de ordenación lineal (f') de los vértices del mismo grafo de la Figura 1.1.a, junto con cada valor $CW_{f'}(v) \, \forall \, v \in V$ en dicha ordenación. En esta ocasión, el valor de la función objetivo viene determinado por *cutwidth* de los vértices C, D y A, siendo $CW_{f'}(G) = 4$.

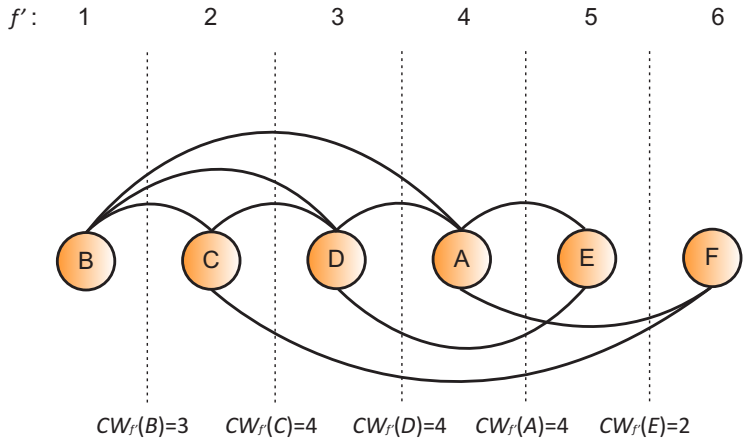


Figura 1.3: Representación gráfica de la ordenación lineal f' de los vértices de G y del valor de $CW_f(v)$ de cada vértice.

El valor óptimo del CMP para un grafo G, sería el menor valor de la función objetivo para todas las posibles ordenaciones de los vértices de G, y quedaría definido de la siguiente manera:

$$CW(G) = \min \{CW_f(G) \ \forall \ f \in \Pi_n\}$$

siendo Π_n el conjunto de todas las posibles ordenaciones de los vértices de G.

1.2.3. Aplicaciones del problema

El CMP es un problema con interés práctico en distintas disciplinas científicas. Sus primeras aplicaciones se remontan a los años setenta [1, 117] teniendo vigencia en la actualidad, como demuestra una reciente patente [153]. A continuación, se describen algunas de sus aplicaciones más relevantes:

- **Migración de redes de telecomunicaciones** [3, 2, 153]: en ocasiones es necesario migrar un conjunto de estaciones de trabajo que están interconectadas formando una red, a otra red más moderna. Para ello, las estaciones son migradas de una en una, de manera que todo el tráfico originado en una estación antigua se reconduce a una estación de la nueva red. El objetivo será establecer un orden de migración de las estaciones, de modo que se minimice el número de conexiones simultáneas que existen entre la red antigua y la nueva, durante el proceso de migración.
- **Diseño de circuitos VLSI** [1, 35, 40, 105, 117, 150]: para los circuitos con gran escala de integración (VLSI, del inglés *Very Large Scale Integration*) se requiere establecer el número de canales necesarios para colocar una serie de dispositivos y sus interconexiones en el circuito. En función de cómo se coloquen dichos dispositivos, se necesitarán más o menos canales para interconectarlos. El número de canales, a su vez, dará una estimación del espacio físico necesario en el circuito para incluir esos dispositivos.

Otras aplicaciones documentadas del problema incluyen: la **representación automática de grafos** [135, 165], el estudio de la **fiabilidad en redes** [93] o la **recuperación de información en hipertextos** [25] entre otras.

1.3. Hipótesis y objetivos

A continuación, se formula la hipótesis que se establece como base de la investigación de esta Tesis Doctoral, así como los distintos objetivos que se pretenden alcanzar durante su desarrollo.

Hipótesis

Una vez identificado el problema que se pretende resolver, el desarrollo de una investigación prosigue con la formulación de una hipótesis inicial. Dicha hipótesis es una propuesta tentativa

que pretende formular una solución al problema abordado. La hipótesis representa un elemento fundamental en el proceso de investigación, ya que sirve de guía en el mismo.

La hipótesis que se plantea para el desarrollo de esta Tesis Doctoral se puede resumir en los siguientes términos: el «*problema de la minimización de la anchura de corte en ordenaciones lineales*» es un problema NP-Difícil con interés práctico en distintas disciplinas científicas e ingenieriles. Por ello, tiene interés el desarrollo de algoritmos que sean capaces de resolverlo de forma eficiente, obteniendo soluciones óptimas o, en su defecto, de alta calidad, en el menor tiempo posible. Para la resolución de este tipo de problemas son relevantes tanto los algoritmos de resolución heurística, como los algoritmos de resolución exacta. Los primeros, son capaces de obtener soluciones de alta calidad en un tiempo razonable, proporcionando así cotas superiores al problema, dado que se trata de un problema de minimización. Por otro lado, los algoritmos de resolución exacta, son capaces de resolver el problema de manera óptima, cuando el tamaño de la instancia es relativamente pequeño. Además, algunas técnicas exactas (como Ramificación y Acotación) son capaces de proporcionar cotas inferiores cuando no consiguen resolver la instancia de manera óptima, en un tiempo determinado.

Los algoritmos heurísticos propuestos harán uso de técnicas metaheurísticas, las cuales se han mostrado como procedimientos eficaces a la hora de abordar problemas de optimización. En particular, las metaheurísticas poblacionales constituyen una subfamilia de este tipo de técnicas, caracterizada por considerar más de una solución de manera simultánea y proporcionar mecanismos de combinación entre ellas. Búsqueda Dispersa [100] (SS, del inglés *Scatter Search*) es un claro exponente de este tipo de metaheurísticas. Otra subfamilia de metaheurísticas son las denominadas metaheurísticas trayectoriales, que parten de una solución inicial y son capaces de generar una trayectoria en el espacio de soluciones. Búsqueda Tabú [73] (TS, del inglés *Tabu Search*) es un ejemplo de metaheurística trayectorial. Por último, las metaheurísticas multiarranque están basadas en la idea de construir y mejorar soluciones durante un número determinado de iteraciones. *Greedy Randomized Adaptive Search Procedure* (GRASP) [56] es un ejemplo de metaheurística multiarranque donde, en cada iteración, la construcción y mejora de la solución puede asemejarse al comportamiento de una metaheurística trayectorial.

En cuanto al desarrollo de algoritmos exactos se refiere, la técnica de Ramificación y Acotación es un método algorítmico general, diseñado para encontrar soluciones óptimas a problemas de Optimización Combinatoria. Este método permite definir diferentes estrategias para recorrer el árbol de exploración (ramificación), y combinarlas con la aportación de cada una de las cotas propuestas, evitando así explorar ciertas partes del árbol (acotación).

Objetivos

El objetivo principal de esta Tesis Doctoral se deriva directamente de la hipótesis enunciada anteriormente y consiste en desarrollar algoritmos de resolución para el «*problema de la*

minimización de la anchura de corte en ordenaciones lineales», tanto desde el punto de vista heurístico como desde el punto de vista exacto.

El algoritmo heurístico propuesto se complementará con las técnicas metaheurísticas que mejor se adecúen al problema, en particular se considerarán GRASP y SS. Por otro lado, el algoritmo exacto hará uso del esquema de Ramificación y Acotación (B&B, del inglés *Branch & Bound*).

Para alcanzar el objetivo principal descrito, es necesario cubrir los siguientes objetivos parciales:

- **Conocer el estado del arte del problema**: se debe hacer un estudio bibliográfico del problema, recopilando las diferentes técnicas utilizadas para la resolución del mismo, tanto con algoritmos heurísticos como con algoritmos exactos. Además, se recogerán los conjuntos de instancias sobre los cuales hayan sido evaluados dichos algoritmos.

- **Diseñar y desarrollar un algoritmo heurístico para la resolución del problema**: para ello se hará uso de técnicas metaheurísticas. En concreto, el trabajo se centrará en el desarrollo de un algoritmo basado en GRASP y SS, incluyendo algoritmos constructivos, de mejora y de combinación de soluciones.

- **Validar el algoritmo heurístico**: se debe comprobar que las soluciones proporcionadas por el algoritmo propuesto son factibles. Además es deseable verificar que cada uno de los componentes que forman parte del algoritmo reflejan las ideas propuestas y tienen aportación al esquema final.

- **Comparar experimentalmente el algoritmo propuesto, con los algoritmos del estado del arte**: se deben identificar los algoritmos del estado del arte que aborden la resolución del problema, desde un punto de vista heurístico, y realizar una comparación exhaustiva entre las diferentes propuestas sobre un conjunto de instancias de referencia.

- **Diseñar y desarrollar un algoritmo exacto para la resolución del problema**: para el desarrollo de un algoritmo exacto se empleará la técnica de Ramificación y Acotación, siendo necesaria la proposición de una serie de cotas inferiores para el problema.

- **Validar el algoritmo exacto**: se debe comprobar que las soluciones proporcionadas por el algoritmo son óptimas, realizando evaluaciones sobre instancias cuyo óptimo es conocido o comparándolas con las obtenidas por un *solver*[1] empleando una formulación del problema. Además se debe evaluar la aportación de las diferentes cotas propuestas.

- **Comparar experimentalmente el algoritmo exacto propuesto con el estado del arte**: se llevará a cabo una comparación del algoritmo de Ramificación y Acotación

[1] *Software* que dispone de algoritmos genéricos para la resolución de modelos de optimización

propuesto, con los algoritmos del estado del arte o, en su defecto, con la formulación entera del problema empleando un *solver* de uso general.

- **Someter los resultados parciales del trabajo de investigación a procesos de revisión por parte de instituciones independientes**: los resultados de la investigación serán enviados a congresos y revistas de reconocido prestigio nacional e internacional en el área, para su posible publicación.

- **Redactar una memoria de investigación**: finalmente, se redactará la memoria de la Tesis Doctoral donde se describirá el problema a resolver, se resumirá el estado del arte del mismo, se detallarán los algoritmos propuestos y se expondrán tanto las aportaciones, como los resultados obtenidos.

1.4. Propuesta algorítmica

En esta sección se aborda de forma más detallada el contenido de la propuesta a desarrollar en esta Tesis Doctoral. En particular se propone la resolución del «*problema de la minimización de la anchura de corte en ordenaciones lineales*» empleando técnicas de resolución exacta y técnicas de resolución heurística.

1.4.1. Técnicas de resolución exacta

Ramificación y acotación es un método general de búsqueda con retroceso conocido por su nombre en inglés como «*Branch & Bound*». El método fue propuesto por A. H. Lang y A. G. Doig en 1960 para programación lineal [101], y es aplicado comúnmente a problemas de optimización con restricciones para la obtención de soluciones óptimas.

El procedimiento de búsqueda consiste en enumerar cada una de las soluciones que componen el espacio de soluciones del problema, organizándolo a modo de árbol. En la Figura 1.4 se muestra un ejemplo de árbol de exploración para un grafo de cuatro vértices, en el que las posibles soluciones son ordenaciones de los vértices del grafo.

La raíz del árbol de exploración representa la solución vacía, cada hoja es una solución completa y cada nodo intermedio una solución parcialmente construida. Durante la exploración del árbol, se ramifican, paso a paso, los nodos que resulten más prometedores; es decir, aquéllos que se espera puedan conducir a una solución mejor que la que se haya encontrado hasta el momento. Para determinar si un nodo es prometedor, se emplean las denominadas funciones de cota. Las cotas son estimaciones acerca de lo prometedor que puede llegar a ser un nodo del árbol de exploración. Cuando se está explorando un nodo, se generan todos los hijos del nodo en curso antes de que cualquier otro nodo pase a ser el nuevo nodo en curso. Por lo tanto, es común que existan nodos que hayan sido generados y estén esperando a ser explorados. A

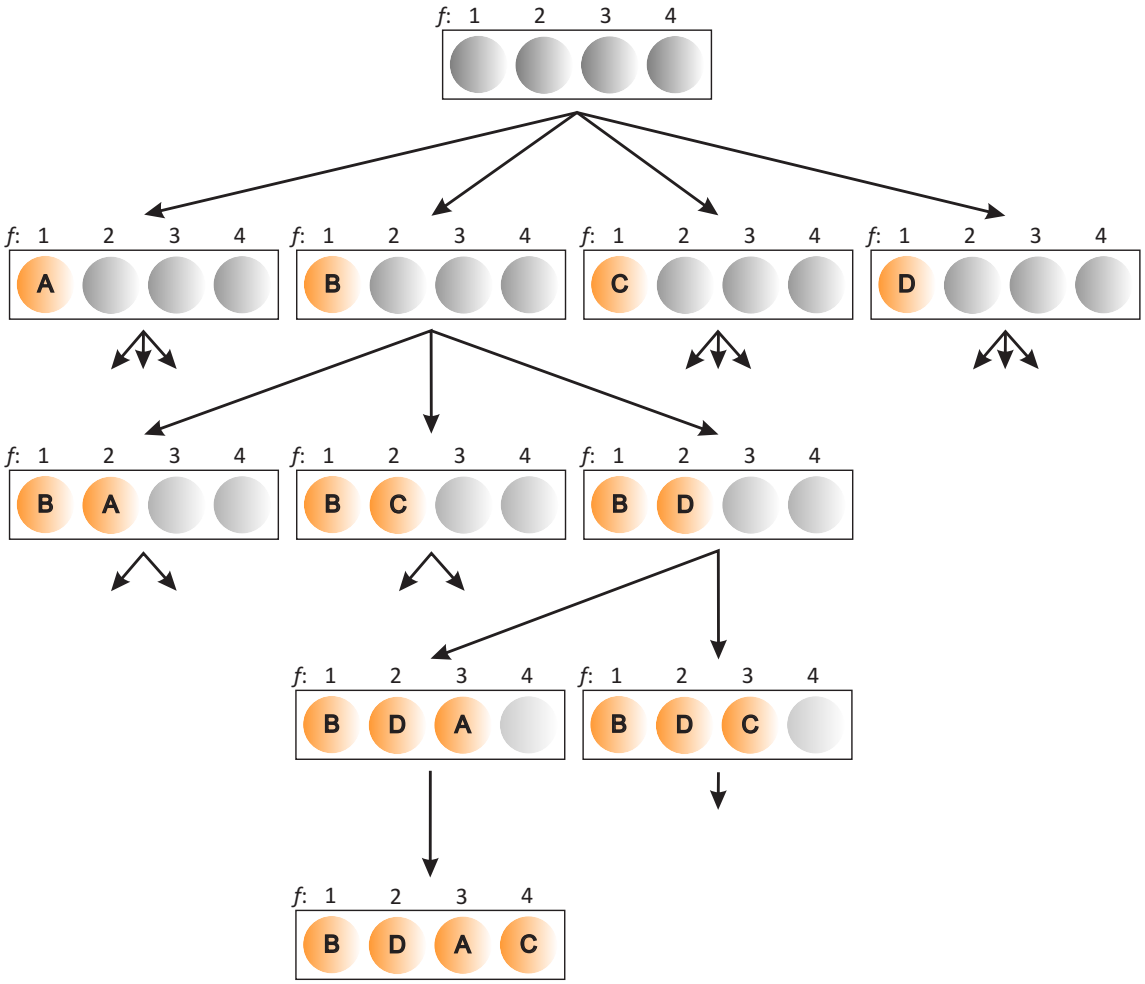


Figura 1.4: Ejemplo de árbol de exploración de un grafo de 4 vértices.

estos nodos se los conoce como nodos vivos y deben ser almacenados en una estructura de datos auxiliar. Cuando se haya terminado de explorar un nodo, se pasará a explorar el nodo vivo más prometedor de todos y así sucesivamente hasta que todos los nodos hayan sido explorados o se haya alcanzado algún criterio de parada preestablecido como, por ejemplo, un tiempo máximo.

Si durante el proceso de búsqueda se encuentra un nodo que no resulta prometedor, es decir, la cota de ese nodo es mayor o igual que la mejor solución encontrada hasta el momento, éste se poda o satura, ya que la cota da una estimación de la mejor solución que se podría llegar a encontrar ramificando ese nodo. Podar un nodo significa evitar explorar las soluciones que pudieran ser alcanzables a través de él, debido a que no conduce a soluciones de mejor calidad. Implícitamente, al podar un nodo, se están podando todos sus hijos en el árbol de exploración.

Para abordar la resolución exacta del CMP se propone la utilización de la técnica de Ramificación y Acotación. Para ello será necesario proponer distintas cotas para el problema, criterios de recorrido del árbol de exploración y estudiar las estructuras de datos más adecuadas para almacenar los nodos vivos del árbol.

1.4.2. Técnicas de resolución heurística

Se propone la utilización de técnicas de resolución heurística para encontrar soluciones aproximadas al «*problema de la minimización de la anchura de corte en ordenaciones lineales*». En concreto, se emplearán técnicas metaheurísticas para obtener resultados de alta calidad en tiempos de cómputo razonables. No obstante, estas técnicas presentan la desventaja de no poder certificar si las soluciones encontradas son óptimas. A continuación, se describen brevemente las heurísticas y metaheurísticas que se plantean para la resolución del problema y que serán detalladas en el Capítulo 4.

Heurísticas

Según el Diccionario de la Lengua Española [152], la etimología de la palabra «heurística», de origen griego, significa: «*Hallar, inventar*» y entre sus definiciones se pueden encontrar las siguientes: «*Técnica de la indagación y del descubrimiento*» y «*En algunas ciencias, manera de buscar la solución de un problema mediante métodos no rigurosos, como por tanteo, reglas empíricas, etc.*». La popularización del concepto «heurística» en el ámbito de la investigación se debe al matemático George Pólya, quien la empleó por primera vez en su libro «*How to solve it*» [148].

En optimización, el término heurística hace referencia a procedimientos capaces de proporcionar soluciones a problemas de optimización. Dichos procedimientos pueden ser genéricos o específicos. Los procedimientos genéricos suelen ser sencillos, adaptables y robustos, además de ser independientes del problema para el que se emplean. Los procedimientos específicos, suelen ofrecer soluciones de más calidad que los genéricos, si bien necesitan ser particularizados para el problema en cuestión.

Una heurística puede partir de una solución previa, o bien, puede construir una solución desde el principio. Las heurísticas que construyen soluciones son denominadas heurísticas constructivas, mientras que las heurísticas que tratan de buscar una mejor solución, realizando movimientos a partir de una solución ya construida, se denominan heurísticas de búsqueda o búsquedas locales. De manera más detallada:

- **Constructivo:** procedimiento heurístico capaz de crear una solución para un problema determinado, partiendo de una solución vacía. Estas heurísticas van añadiendo elementos a la solución, paso a paso, hasta que la solución es completada. En cada iteración, un elemento es escogido empleando un criterio de selección. Algunos de los criterios de selección más utilizados son:

  - **Estrategia voraz**: también conocida como miope o ávida, selecciona a cada paso el elemento más prometedor para la solución parcial en ese momento, con independencia

de lo que pase en el futuro.

- **Estrategia de descomposición**: consiste en subdividir el problema en problemas más pequeños de manera recursiva, hasta que el tamaño del problema abordado resulte trivial. Después, se combinan las soluciones obtenidas para esos subproblemas, formando una solución final.
- **Estrategia de reducción**: consiste en reducir el espacio de búsqueda mediante la identificación de características comunes en soluciones de alta calidad, para introducirlas en la solución construida, asumiendo que la solución óptima también tendrá dichas características.
- **Estrategias de manipulación del modelo**: basada en la simplificación del modelo del problema y la obtención de una solución para el modelo simplificado. Una vez obtenida esta solución, se extrapola al problema original. Algunos métodos destacados de este tipo de estrategias son la linealización, la agrupación de variables o la introducción de nuevas restricciones.

- **Búsqueda local:** también denominado método de mejora, es un procedimiento heurístico que parte de una solución ya creada y trata de encontrar una solución de mejor calidad. Está basado en el que probablemente sea uno de los procedimientos de optimización más antiguos: prueba y error [142]. Para ello, realiza modificaciones o movimientos dentro de la solución de partida y evalúa la calidad de la nueva solución obtenida. La heurística finaliza cuando ha agotado todos los posibles movimientos a realizar y ninguno de ellos ha resultado en una mejora de la solución de partida. El movimiento a realizar puede seguir distintas estrategias:

  - Estrategia *Best Improvement*: consiste en evaluar todos los posibles movimientos de una solución particular, y realizar el movimiento que suponga una mejora mayor en la función objetivo.
  - Estrategia *First Improvement*: consiste en evaluar uno a uno los posibles movimientos de una solución y realizar el primer movimiento que suponga una mejora de la función objetivo.

Las técnicas heurísticas presentan la desventaja de que, en general, son incapaces de escapar de óptimos locales. En optimización, se denomina «óptimo local» a una solución que es la mejor de su vecindad, entendiendo por vecindad de una solución, el conjunto de soluciones alcanzables desde dicha solución al realizar un movimiento, pero que no es necesariamente el óptimo global.

En esta Tesis Doctoral se propone el desarrollo de heurísticas, tanto constructivas como de búsqueda local, para la resolución del «*problema de la minimización de la anchura de corte en ordenaciones lineales*».

Metaheurísticas

Como se ha mencionado anteriormente, el principal inconveniente que presentan las heurísticas, en el área de la optimización, es que no son capaces de escapar de óptimos locales. A raíz de esto, se define un nuevo tipo de técnicas, las «metaheurísticas». Según el Diccionario de la Lengua Española [152] el prefijo «meta-» significa «*"Junto a", "después de", "entre", "con" o "acerca de"*», si bien es común darle el sentido de «más allá de». En inglés, está aceptado también el uso en sustantivo compuesto «*por encima de*» [139]. El término «metaheurística» como tal, fue acuñado por Fred Glover en el año 1986 [68]. Con este término, pretendía definir un procedimiento maestro de alto nivel que guiase y modificase otras heurísticas para explorar soluciones más allá de los óptimos locales.

A continuación, se repasan las metaheurísticas consideradas para abordar el CMP junto con las heurísticas anteriormente descritas: GRASP y Búsqueda Dispersa.

Greedy Randomized Adaptive Search Procedure

GRASP es una metaheurística de propósito general, cuyo acrónimo proviene de *Greedy Randomized Adaptive Search Procedure* y que podría ser traducido como «*Procedimientos de búsqueda voraces, aleatorizados y adaptativos*» [154]. Fue introducida por Tom Feo y Mauricio Resende en [55], aunque no fue hasta [56] cuando adquirió la terminología y forma definitiva actual.

Esta metaheurística se podría clasificar como un procedimiento multiarranque, en el que cada arranque se corresponde con una iteración del algoritmo que, a su vez, consta de dos fases principales: construcción y mejora.

Las construcciones se llevan a cabo con un procedimiento heurístico constructivo que cumple, a la vez, con tres de las características que dan nombre al método: voraz, aleatorio y adaptativo. A la hora de construir una solución, se van añadiendo elementos a la misma de forma iterativa. Típicamente, el elemento elegido a cada paso se escoge de manera aleatoria, de entre un subconjunto de elementos que previamente han sido escogidos de manera voraz. Es decir, en lugar de escoger el elemento más prometedor a cada paso, GRASP introduce cierta aleatoriedad en el proceso, seleccionando uno de los mejores. La introducción de aleatoriedad en la fase de construcción diversifica la búsqueda cuando se realizan varias construcciones, permitiendo así explorar áreas distintas del espacio de soluciones. El procedimiento es adaptativo ya que, cuando se tiene que seleccionar un nuevo elemento, se revalúa el nuevo subconjunto de elementos prometedores, que no tienen por qué ser los mismos de la iteración anterior, ya que la solución ha cambiado.

La fase de mejora se aplica directamente sobre la solución obtenida después de cada construcción. Dicha mejora podría estar conducida por una búsqueda local o bien por otra

metaheurística, algo que convierte a GRASP en un procedimiento muy atractivo. Tras realizar el número de construcciones y mejoras indicado, el algoritmo devuelve la mejor solución encontrada.

En la Figura 1.5 se muestran las principales etapas de un diseño básico de GRASP. Como se puede apreciar en la figura, el procedimiento comienza inicializando la mejor solución encontrada, la cual será actualizada a lo largo del proceso de búsqueda. En cada iteración se realiza una construcción GRASP y una mejora de la solución construida. Si la nueva solución obtenida es mejor que la mejor solución hasta ese momento, ésta se actualiza. El proceso se repite el número de iteraciones predefinidas.

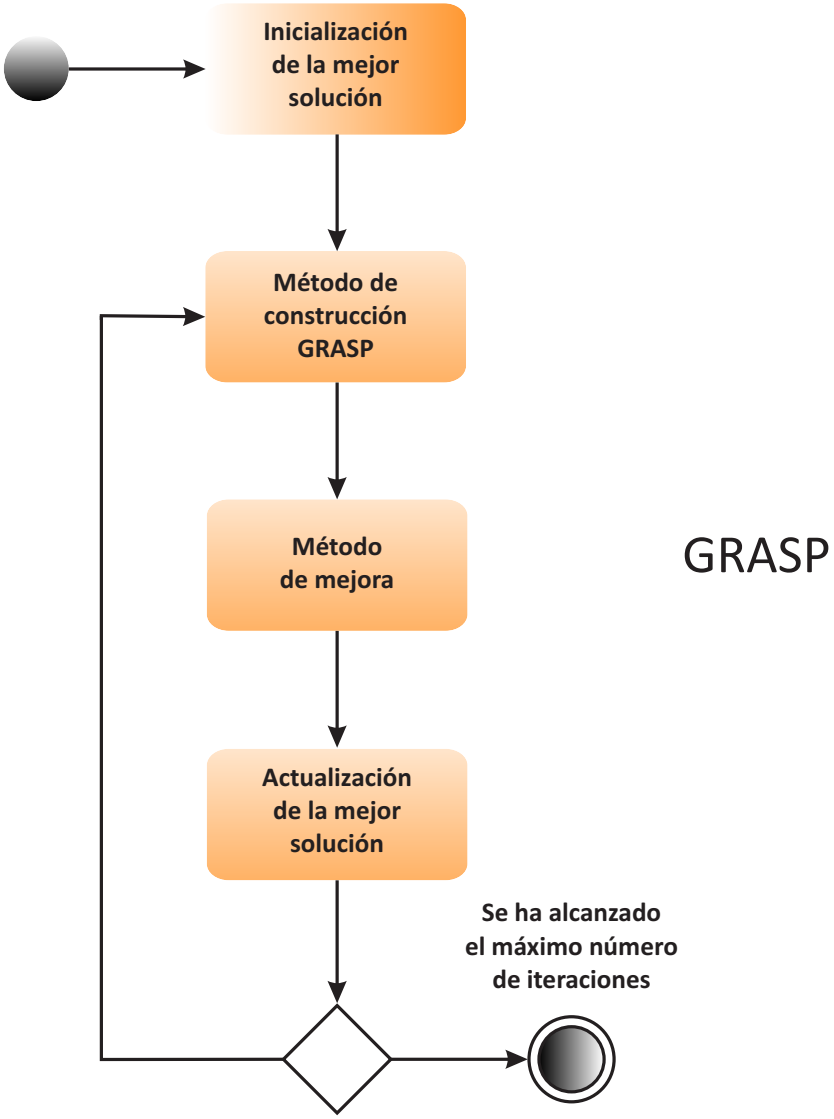


Figura 1.5: Representación esquemática de un diseño básico de GRASP.

Búsqueda Dispersa

Búsqueda Dispersa (SS, del inglés *Scatter Search*) es un método metaheurístico evolutivo que ha sido aplicado con éxito a un gran número de problemas de optimización. La primera descripción del método fue publicada por Fred Glover en [67] aunque las bases del esquema actual se sentaron en [71]. Se basa en la combinación de un conjunto de soluciones, denominado conjunto de referencia, sobre el que se realiza una exploración sistemática. La combinación de soluciones permite diversificar la búsqueda mientras que la búsqueda local, aplicada sobre las soluciones combinadas, la intensifica. Desde un punto de vista algorítmico, SS está compuesto por cinco procedimientos que son descritos a continuación:

- **Generación de soluciones diversas:** el método SS debe partir de un conjunto inicial de soluciones, que pueden ser generadas empleando una heurística constructiva. Está bien documentado (véase por ejemplo [100]) que las soluciones iniciales en SS deben tener una calidad razonablemente buena, pero también ser suficientemente diversas. Es decir, deben encontrarse dispersas en el espacio de soluciones para permitir a la búsqueda local y al método de combinación alcanzar diferentes óptimos locales. El número de soluciones inicial puede variar según la implementación, aunque típicamente se utilizan 100 soluciones de partida.

- **Método de mejora:** como ya se ha indicado anteriormente, los métodos de mejora (o búsqueda local) permiten encontrar soluciones de mejor calidad, a partir de una solución ya construida, explorando la vecindad de dicha solución. Dado que la estructura de una solución para el CMP constituye una ordenación, el mecanismo para encontrar soluciones de mejor calidad consiste en realizar movimientos entre los elementos de la misma. Aunque es posible definir otros movimientos, en general, existen dos estrategias a la hora de realizar movimientos en soluciones de este tipo: intercambios e inserciones.

  - **Intercambios:** los movimientos basados en intercambios consisten en seleccionar dos vértices de la ordenación e intercambiar sus posiciones. Es necesario, por lo tanto, determinar qué vértices se desea intercambiar.
  - **Inserciones:** los movimientos basados en inserciones consisten en seleccionar un vértice de la ordenación e insertarlo en una posición determinada. Para ello, es necesario determinar qué vértice se desea mover y la posición a la que moverlo.

- **Creación/Actualización del conjunto de referencia:** el conjunto de referencia (*RefSet*, del inglés *Reference Set*) es un subconjunto de soluciones creado a partir del conjunto inicial descrito anteriormente. Las soluciones que forman el *RefSet* se seleccionan empleando dos criterios: calidad y diversidad. Un porcentaje de soluciones del *RefSet* es escogido por calidad y las restantes por ser soluciones con una estructura diferente

a las primeras, aunque su calidad no sea tan buena. Esto es lo que se conoce como introducir diversidad en el conjunto. Además de la formación inicial del *RefSet*, éste debe ser actualizado a lo largo de la ejecución del método SS. Como resultado de los procedimientos de combinación y mejora, se obtienen nuevas soluciones, que pueden pasar a formar parte del *RefSet*. Para ello, típicamente se emplean criterios de calidad, aunque también es posible considerar un balance entre calidad y diversidad. El tamaño habitual del *RefSet* es de 10 soluciones, con un porcentaje inicial del 50 % de soluciones escogidas por calidad y 50 % por diversidad, aunque estos parámetros pueden ser modificados.

- **Generación de subconjuntos:** como ya se ha indicado, SS se basa en la combinación exhaustiva de soluciones. Las soluciones a combinar son aquéllas presentes en el *RefSet*, en cada iteración del algoritmo. Para realizar una combinación entre dos o más soluciones es necesario generar los subconjuntos a combinar. La opción más sencilla y habitual consiste en formar subconjuntos de dos soluciones cada uno.

- **Combinación de soluciones:** consiste en construir una solución (solución combinada) a partir de dos o más soluciones del *RefSet* (denominadas soluciones de referencia) que han sido escogidas mediante el método de generación de subconjuntos anteriormente descrito. Los mecanismos de combinación difieren en función del tipo de solución que se desee combinar. En particular, cuando la estructura de la solución es una ordenación (como es el caso del CMP) es común utilizar el denominado mecanismo de combinación por votos [69].

La interacción entre los métodos anteriormente descritos se muestra en la Figura 1.6. Inicialmente, el método de generación de soluciones diversas creará un conjunto de soluciones. Dichas soluciones serán la entrada al método de mejora. Con las soluciones ya mejoradas, el método de actualización del conjunto de referencia evaluará las soluciones candidatas a formar parte del mismo. A partir del conjunto de referencia se generarán subconjuntos de soluciones que serán combinadas utilizando el método de combinación de soluciones. Estas soluciones serán nuevamente mejoradas y se volverán a evaluar como candidatas a formar parte del conjunto de referencia. A priori, el algoritmo termina cuando no se añaden nuevas soluciones al *RefSet*, aunque es posible realizar una reconstrucción del mismo.

En esta Tesis Doctoral se propone la combinación de las metaheurísticas GRASP y SS para el diseño e implementación de un algoritmo para el «*problema de la minimización de la anchura de corte en ordenaciones lineales*».

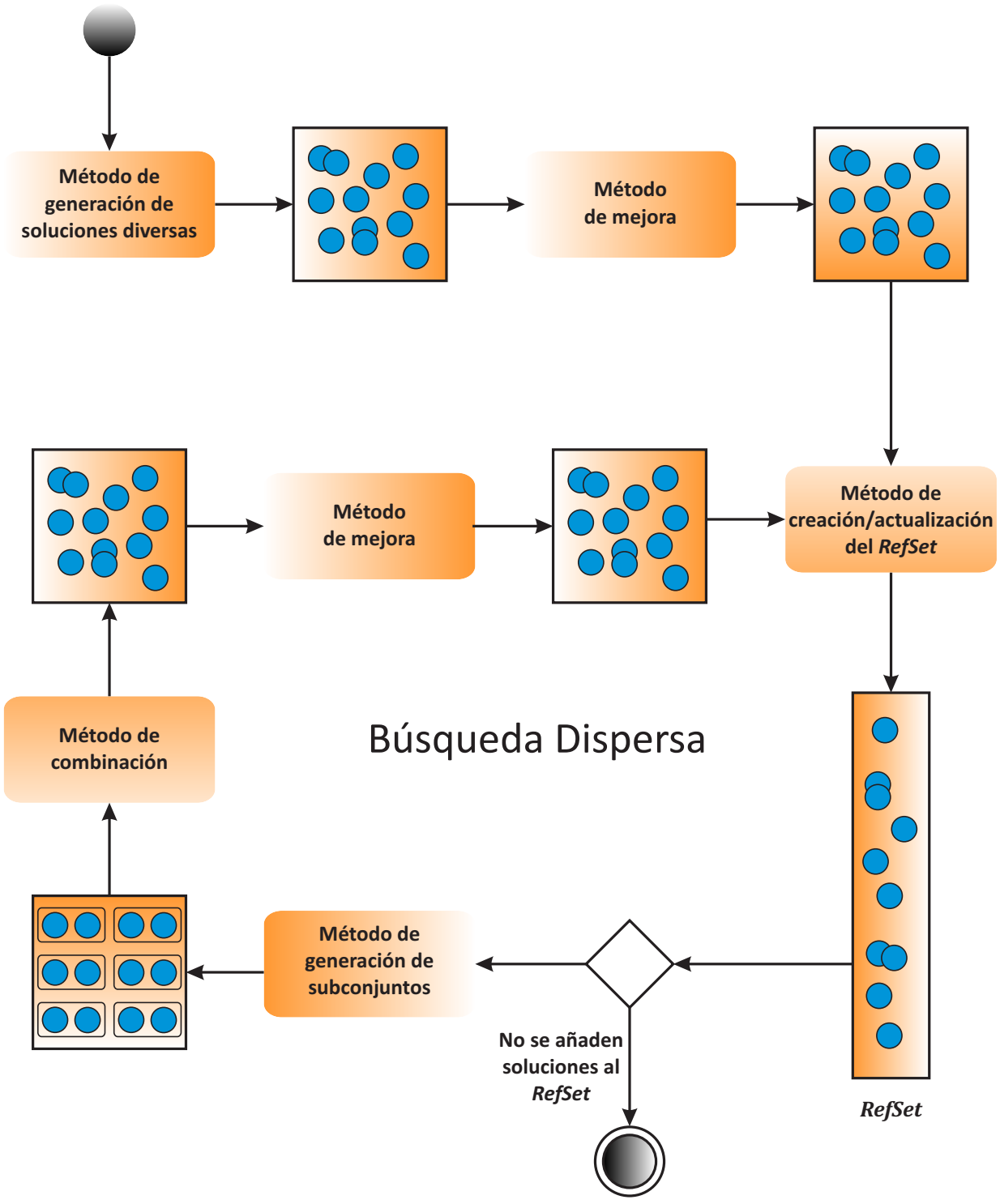


Figura 1.6: Representación esquemática de un diseño básico de SS (adaptado de [100]).

1.5. Método de investigación

El proceso de investigación en optimización sintetiza los pasos básicos a seguir en el desarrollo de una investigación científica en el área. Dicho proceso presenta similitudes con otras áreas, pero también características específicas propias de la optimización, que deben ser consideradas con especial atención.

Una vez identificado el problema de optimización que se pretende resolver, el proceso comienza con una revisión del estado del arte del mismo. Como resultado de dicha revisión se identifican los algoritmos desarrollados para el problema en cuestión, así como el conjunto de instancias empleadas en su evaluación. De acuerdo con el estudio realizado, se plantea una hipótesis para la resolución del problema, que desembocará en la codificación de uno o varios algoritmos específicos. La validación de los algoritmos se realiza en distintas etapas. En una primera etapa, se comprueba que el código no contiene errores. A continuación, se evalúa la contribución de los elementos integrantes del algoritmo y, por lo tanto, la validez de las estrategias propuestas. Si el resultado de las validaciones anteriores es positivo, se realiza una validación final de la propuesta planteada, comparando los resultados obtenidos con los de otros algoritmos presentes en el estado del arte. Para ello, se emplea un conjunto de instancias de referencia que permitan comparar los algoritmos entre sí. El proceso descrito es iterativo, ya que es común que, tras llevar a cabo las distintas validaciones, sea necesario volver a la etapa de desarrollo o incluso modificar la hipótesis inicial.

El proceso de investigación descrito se muestra, de manera gráfica, en el diagrama de actividad de la Figura 1.7. Como se puede observar en dicha figura, si los resultados finales satisfacen la hipótesis inicial, el proceso concluye con la publicación de la misma junto con los resultados experimentales.

Algunas necesidades derivadas las etapas más importantes del proceso de investigación son:

- **Estado del arte**: el resultado de esta etapa será la identificación de un conjunto de algoritmos, artículos e instancias relevantes para el problema que se pretende resolver. Es necesario, por lo tanto, determinar una manera eficiente de almacenar y acceder a dicha información.

- **Desarrollo**: la etapa de desarrollo consumirá una gran parte del tiempo empleado en el proceso de investigación. Esta etapa puede beneficiarse de la utilización de herramientas tales como entornos de desarrollo o repositorios de código. Asimismo, muchos esquemas algorítmicos son reutilizados una y otra vez, por lo que el disponer de plantillas con las partes comunes ya implementadas puede facilitar la tarea al desarrollador.

- **Validación**: para esta etapa serán necesarios mecanismos de ejecución de algoritmos, tanto en local como en remoto, así como herramientas para el análisis de los resultados

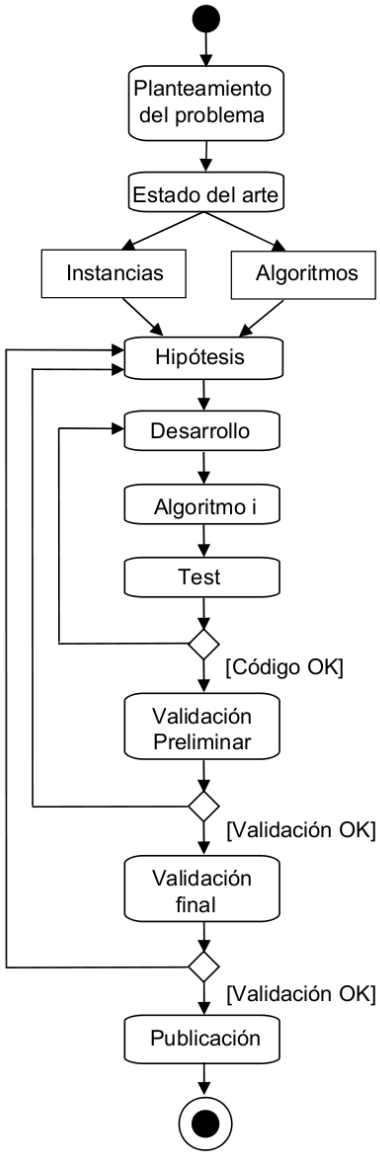


Figura 1.7: Diagrama de actividad del proceso de investigación en el área de optimización.

obtenidos. El diseño de la experimentación es una tarea compleja que se ve beneficiada de la definición de procesos sistemáticos a la hora de realizar experimentos. Por su parte, los análisis llevados a cabo suelen estar dirigidos por estadísticos bien conocidos, por lo que la utilización de librerías que incluyan los análisis más comunes, puede resultar de gran utilidad.

- **Difusión de resultados/Publicación**: durante la etapa de difusión de resultados, debe ser posible consultar nuevamente la información recogida en la etapa del estado del arte. Además, serán necesarios soportes auxiliares, como por ejemplo una página web, que haga accesibles instancias y mejores resultados encontrados.

1.6. Medios utilizados

La realización de este trabajo de investigación se ha llevado a cabo dentro del grupo de investigación Gavab, compuesto por profesores que pertenecen al área de Ciencia de la Computación e Inteligencia Artificial, encuadrados en el Departamento de Ciencias de la Computación de la Escuela Técnica Superior de Ingeniería Informática de la Universidad Rey Juan Carlos. Además, durante el proceso de esta investigación, se mantiene estrecha colaboración con el Departamento de Estadística e Investigación Operativa de la Facultad de Matemáticas de la Universidad de Valencia. Para la realización de esta Tesis Doctoral se cuenta, entre otros, con los siguientes medios:

- Laboratorio de investigación del grupo Gavab
- Biblioteca de la Universidad Rey Juan Carlos, con acceso a préstamo interbibliotecario, Consorcio Madroño, bases de datos especializadas, acceso a las bibliotecas digitales de IEEE y ACM, etc.
- *Hardware* provisto por el grupo de investigación (ocho computadores Intel Core 2 Quad CPU con 6 GB de RAM) y *hardware* provisto por la Universidad (computador Intel Core 2 Duo con 2 GB de RAM).

1.7. Estructura de la memoria

Para finalizar el capítulo introductorio de esta Tesis Doctoral, se describe de forma resumida la estructura del resto de la memoria, incluyendo una breve descripción de cada capítulo presente en el documento:

- **Capítulo 2**: en el Capítulo Segundo se realiza un estado del arte del «*problema de la minimización de la anchura de corte en ordenaciones lineales*». En él se recogen y clasifican las publicaciones más relevantes relacionadas con el CMP. Además, se hace especial hincapié en aquellos trabajos que abordan la resolución del CMP tanto desde el punto de vista exacto como desde el punto de vista heurístico. Para ello, se detalla una posible formulación entera para el problema, propuesta en un trabajo previo, así como los algoritmos heurísticos que abordan su resolución. Se repasan también algunas familias de grafos con óptimo conocido, así como algoritmos polinómicos para algunos tipos de grafos específicos.
- **Capítulo 3**: en el Capítulo Tercero se aborda la resolución del problema desde un punto de vista exacto. Inicialmente, se repasan las diferentes técnicas existentes para la resolución de problemas de optimización de manera exacta. A continuación, se repasan los trabajos

previos relacionados con este enfoque, introducidos en el Capítulo 2, para continuar con la descripción del algoritmo de Ramificación y Acotación propuesto para la resolución del CMP. Para ello, se detallan distintos recorridos del árbol de exploración y se proponen un conjunto de cotas inferiores y un primer algoritmo heurístico, que servirá para calcular cotas superiores iniciales.

- **Capítulo 4**: el Capítulo Cuarto recoge los algoritmos propuestos para la resolución del problema desde un punto de vista heurístico. Inicialmente, se repasan las diferentes técnicas existentes para la resolución de un problema de optimización de manera aproximada. A continuación se describen las heurísticas propuestas para abordar el problema. En concreto se propone un algoritmo basado en la combinación de GRASP y Búsqueda Dispersa, para el que se diseñan varias heurísticas constructivas, de mejora y métodos de combinación de soluciones.

- **Capítulo 5**: en el Capítulo Quinto, se lleva a cabo una extensa evaluación experimental de los algoritmos propuestos, tanto en el Capítulo 3, como en el Capítulo 4. La validación de dichas propuestas se realiza mediante la comparación de los resultados obtenidos por los algoritmos propuestos, con los resultados obtenidos por los algoritmos del estado del arte (en el caso heurístico) y de la formulación entera en combinación con un *solver* genérico (en el caso exacto).

- **Capítulo 6**: por último, en el Capítulo Sexto se presentarán las conclusiones de este trabajo de investigación. En él, se resumirán también las principales aportaciones derivadas del mismo, en forma de publicaciones, y los posibles trabajos futuros.

Además del contenido anteriormente descrito, se incluyen cuatro apéndices en los que se presentan, de manera detallada, los conjuntos de instancias empleados; los resultados obtenidos por los algoritmos propuestos sobre el conjunto de instancias más complejas, instancia por instancia; la librería *Optsicom Framework* empleada en el desarrollo de los algoritmos heurísticos propuestos y un listado de *solver* disponibles.

Capítulo 2

Estado del arte

En este capítulo se repasan los trabajos relacionados con el «problema de la minimización de la anchura de corte en ordenaciones lineales». En él, se recogen los algoritmos heurísticos presentes en el estado del arte, los algoritmos polinómicos para grafos específicos, las cotas conocidas del problema y una posible formulación entera del mismo. No obstante, de acuerdo con el enfoque de esta Tesis Doctoral, sólo se describirán en detalle los algoritmos que abordan el problema desde un punto de vista exacto o desde un punto de vista heurístico, para grafos generales.

2.1. Introducción

La bibliografía relacionada con el CMP es extensa, encontrándose trabajos que tratan el problema desde diferentes perspectivas. Existen trabajos que estudian la complejidad computacional del mismo; otros en los que se realizan relajaciones del problema; aplicaciones prácticas; algoritmos aproximados; algoritmos polinómicos para clases de grafos particulares e incluso una formulación entera para el mismo.

A lo largo de este capítulo se lleva a cabo un recorrido por dichos trabajos. También se resume, a modo de histograma, la aparición cronológica de las publicaciones relacionadas con el CMP. Por último, se detallan las propuestas que abordan el problema empleando un enfoque directamente relacionado con el propuesto en esta Tesis Doctoral.

Los problemas de optimización, en ocasiones, reciben diferentes nombres en las publicaciones presentes en el estado del arte. También, es común identificar otros problemas con nombres parecidos, pero que no se corresponden exactamente con el mismo problema. Cuando se desea trabajar sobre un problema en particular, es de especial relevancia identificar los nombres con los que los distintos autores se han referido a él en sus publicaciones. Esto conduce, a su vez, a identificar los métodos que han sido aplicados para abordar el problema en cuestión. En concreto, se han encontrado referencias al «*problema de la minimización de la anchura de corte en ordenaciones lineales*» con los siguientes nombres:

- *Cutwidth Minimization Problem* ([10, 111, 117, 159])
- *Minimum Cut Linear Arrangement* ([62, 171])
- *Min-Cut Linear Arrangement* ([80, 105, 129, 117, 125])
- *Network Migration Scheduling* ([3, 2, 153])
- *Board Permutation Problem*[1] *o Backplane Ordering* ([37, 38, 39, 125])
- *Folding Number* ([32, 35])
- *Minimal Congestion* ([111, 159])
- *Edge Separation* ([167])
- *Optimal Linear Cut Arrangement* ([64])

2.1.1. Complejidad computacional

Como se ha mencionado en la Sección 1.1, existe una familia de problemas (NP-Completos) en la cual, para ninguno de sus miembros se conoce un algoritmo de resolución en tiempo polinómico, aunque no se ha podido demostrar que no exista. Esta familia, presenta la peculiaridad de que, si uno de ellos pudiera ser resuelto en tiempo polinómico, es decir, si existiera un algoritmo capaz de resolver uno de ellos en tiempo polinómico, entonces todos los problemas de dicha familia serían resolubles en tiempo polinómico.

Los cimientos de la teoría de los problemas NP-Completos fueron puestos por Stephen A. Cook en 1971 quien, en su publicación «*The complexity of theorem-proving procedures*» [41], enfatizó el significado de la expresión «*polynomial time reducibility*», que hace referencia a aquellas reducciones para las cuales las transformaciones necesarias pueden ser ejecutadas en tiempo polinómico. Es decir, si se tiene una reducción en tiempo polinómico de un problema a otro, esto asegura que cualquier algoritmo polinómico para el segundo problema puede ser convertido en su correspondiente algoritmo polinómico para el primer problema. Además, Cook centró su atención en la denominada «clase de problemas de decisión NP». Un problema de decisión es aquél cuya solución es "sí" o "no". Por último y quizá lo más importante, probó que un problema de la clase NP, denominado «*Satisfiability*» (SAT) tiene la propiedad de que cualquier otro problema NP puede ser reducido a él en tiempo polinómico. Si este problema pudiera ser resuelto en tiempo polinómico, todos podrían serlo pero, si cualquier problema de esta familia se demostrase intratable (se dice que un problema es intratable cuando no es resoluble en tiempo polinómico) entonces SAT también sería intratable.

[1] Generalización para hipergrafos

Poco después de la publicación de Cook, Richard Karp presentó una colección de resultados en los que probaba que las versiones de decisión de muchos problemas combinatorios bien conocidos eran, al menos, tan difíciles como SAT [95]. La relación entre los problemas estudiados por Karp, a la hora de reducir unos a otros, se muestra en la Figura 2.1. Entre ellos se encuentra el denominado problema del «*Max-Cut*», a partir del cual se demostró que el «*Simple Max-Cut*» (equivalente al «*Max-Cut*» pero restringiendo el peso de las aristas del grafo a 1) era también NP-Completo [63]. Este último problema («*Simple Max-Cut*») fue relevante poco después en el estudio de la complejidad del CMP.

La demostración de que la versión de decisión del CMP es un problema NP-Completo es obra de otro de los pioneros en el estudio de la complejidad computacional, Larry J. Stockmeyer, quien, en una comunicación privada dirigida a Michael R. Garey y David S. Johnson, realizó una reducción del problema al «*Simple Max-Cut*». No obstante, dicha contribución nunca llegó a ser publicada [64, 62] y no fue hasta 1977 cuando Fanica Gavril recogió dicho resultado [64].

Otros estudios posteriores de la complejidad computacional del CMP condujeron a la demostración de que el CMP es NP-Completo para grafos con grado máximo de tres [118] y para grafos planos con grado máximo de tres [129]. Por otro lado, en [45] se demostró que el CMP permanece NP-Completo restringido a grafos tipo *grid graph*[2] y *unit disk graph*[3].

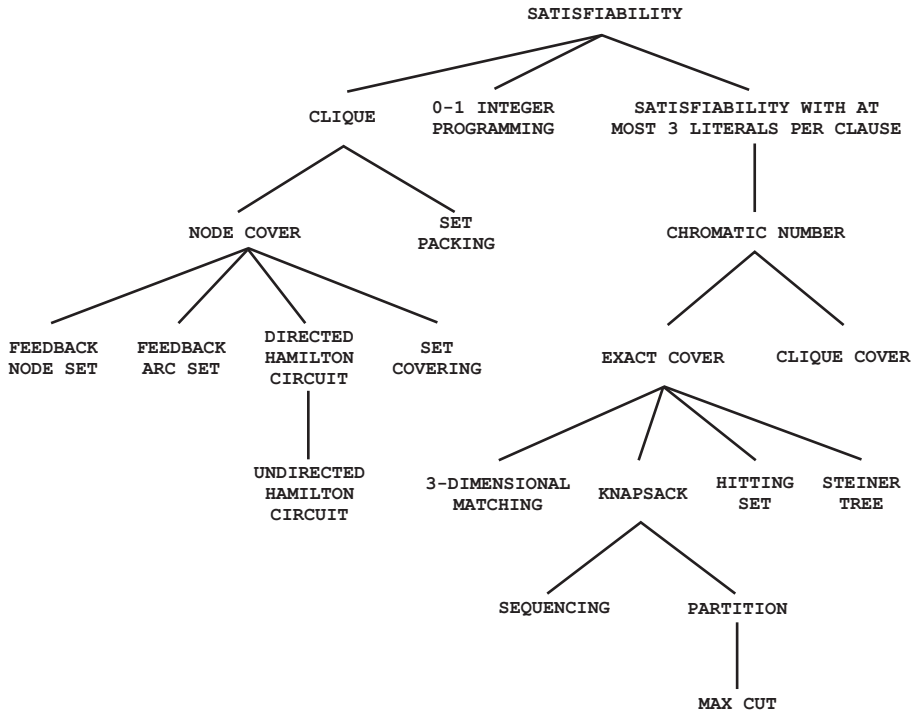


Figura 2.1: Relación existente entre distintos problemas NP-Completos a la hora de reducir unos a otros (Adaptado de [95]). Nótese que se ha respetado el nombre en inglés de los problemas de acuerdo con la figura original.

[2]Subgrafo finito de un grafo tipo *grid* infinito, donde un *grid* es el producto cartesiano de dos caminos [45].

[3]Un grafo se considera un *unit disk graph* si cada vértice puede puede ser asignado a un disco con un diámetro de longitud la unidad, de modo que situados en el plano, cada par de vértices son adyacentes si y sólo si sus correspondientes discos intersectan [45].

2.1.2. Aplicaciones del problema

Tal y como se recoge en el Capítulo 1, el CMP es un problema con aplicaciones prácticas. Los trabajos publicados en este sentido se dividen de acuerdo con la aplicación práctica del mismo. A continuación se repasan dichas aplicaciones:

- Diseño de circuitos VSLI ([1, 35, 40, 105, 117, 150])
- Migración de redes de telecomunicaciones ([3, 2, 153])
- Recuperación de la información ([25])
- Representación automática de grafos ([135, 165])
- Fiabilidad en redes ([93])
- Subrutina para un algoritmo de planos de corte para resolver el problema del TSP ([90])
- Ingeniería de proteínas ([17])

2.1.3. Relación con otros problemas de optimización

El CMP está relacionado con otros problemas de optimización, de manera que unos suponen cotas para otros. A continuación, se repasan las relaciones más relevantes, en las que se hace referencia a los distintos problemas empleando su nombre en inglés, por el que son comúnmente conocidos.

Bandwidth

El *Bandwidth* (BW) de un grafo $G(V, E)$ fue formalmente definido en [84] y su versión de decisión es NP-Completa [141]. Consiste en encontrar una ordenación lineal de los vértices de G, de modo que se minimice la máxima longitud de una arista en la ordenación. Siendo $deg(G)$ el grado máximo de G, la relación existente entre el BW y el CMP (recogida en [117]) se puede expresar en los siguientes términos:

$$CMP(G) \leq \left\lfloor \frac{1}{2}\, deg(G) \right\rfloor \cdot BW(G) + 1$$

Edge Bisection

La versión de decisión del *Edge Bisection* (EB) de un grafo $G(V, E)$ es NP-Completa [63]. El problema consiste en encontrar una ordenación de los vértices de G de modo que, al realizar una partición de los vértices del grafo en dos conjuntos disjuntos del mismo tamaño (o con una

unidad de diferencia, si el número de vértices es impar) se minimice el número de aristas que unen ambos conjuntos. La relación existente entre el EB y el CMP (recogida en [46]) se puede expresar en los siguientes términos:

$$EB(G) \leq CMP(G)$$

Minimum Linear Arrangement

El *Minimum Linear Arrangement* (MinLA) un grafo $G(V, E)$ fue introducido originalmente en [83] y su versión de decisión es NP-Completa [63]. Consiste en encontrar una ordenación de los vértices de G, de modo que la suma de las longitudes de todas sus aristas en la ordenación lineal, sea minimizada. Siendo n el número de vértices del grafo, la relación existente entre el MinLA y el CMP (recogida en [46]) se puede expresar en los siguientes términos:

$$MinLA(G) \leq n \cdot CMP(G)$$

Search Number

El *Search Number* (SN) de un grafo fue introducido en [144]. Se trata de un problema NP-Difícil [121], siendo su versión de decisión NP-Completa [102]. Consiste en, dado un grafo conexo, no dirigido, $G(V, E)$, encontrar el mínimo número de buscadores que es necesario introducir en G para garantizar la captura de un fugitivo quien se puede mover arbitrariamente a través de las aristas de G.

Inicialmente todas las aristas del grafo están contaminadas. Una arista $e = \{x, y\}$ es limpiada si hay un buscador en x y un segundo buscador se mueve de x a y o, si hay un buscador en x y todas las aristas que llevan a x, excepto e, están limpias y el buscador en x se mueve a lo largo de e hasta alcanzar y.

Una arista limpia e resulta contaminada nuevamente, si se produce el movimiento o borrado de un buscador, que crea un camino sin buscadores desde una arista contaminada hasta e.

Una estrategia de búsqueda para G consiste en una secuencia de etapas de búsqueda, que conduzca a que todas las aristas estén limpias simultáneamente. Una etapa de búsqueda es una de las siguientes operaciones:

- colocar a un buscador en un vértice
- mover a un buscador a lo largo de una arista
- eliminar a un buscador de un vértice

El SN de un grafo es, por lo tanto, el mínimo número de buscadores para el que existe una estrategia de búsqueda.

Siendo $deg(G)$ el grado máximo de G, la relación existente entre el SN y el CMP (recogida en [117]) se puede expresar en los siguientes términos:

$$SN(G) \leq CMP(G) \leq \left\lfloor \frac{1}{2}\, deg(G) \right\rfloor \cdot (SN(G) - 1) + 1$$

Además, el SN y el CMP son idénticos con grafos cuyo grado máximo es tres.

2.1.4. Cotas inferiores conocidas

Algunas de las relaciones entre el CMP y otros problemas de optimización, incluidas en la Sección 2.1.3, permiten establecer cotas inferiores para el CMP. Además de éstas, se conocen también otras cotas inferiores al problema basadas en propiedades espectrales y en cortes fundamentales. A continuación, se describe cada una de ellas.

Cotas basadas en propiedades espectrales

Dado un grafo $G(V, E)$ con $|V| = n$, se define su matriz Laplaciana (L_G) como una matriz de tamaño $n \times n$, donde $\forall\, u, v \in V$ se tiene:

$$L_G[u, v] = \begin{cases} -1 & \text{si } uv \in E \\ 0 & \text{si } uv \notin E \\ grado(u) & \text{si } u = v \end{cases}$$

Por construcción L_G es semidefinida positiva. Por lo tanto, tiene n autovalores reales no negativos, tales que $0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$. La secuencia $\lambda_1, \lambda_2, \ldots, \lambda_n$ es conocida como el espectro del grafo G. Si G es un grafo conexo, se cumple que $0 = \lambda_1 < \lambda_2$ [127].

Suponiendo que G es un grafo conexo con n vértices y que λ_2 es el segundo autovalor más pequeño de la matriz Laplaciana de G, entonces se cumple [92]:

$$CMP(G) \geq \frac{\lambda_2 \left\lfloor \frac{1}{2}\, n \right\rfloor \left\lceil \frac{1}{2}\, n \right\rceil}{n}$$

Cotas basadas en cortes fundamentales

Sea $G(V, E)$ un grafo con $|V| = n$ y sean s y t dos vértices distintos de G, denominados fuente y destino respectivamente. El conocido teorema del «*max-flow min-cut*» [58] establece que el máximo valor del flujo entre s y t es igual al mínimo corte de aristas separando s y t. Dado que existen algoritmos eficientes para computar este corte mínimo, y que existen $\frac{n \cdot (n-1)}{2}$

posibles elecciones para s y t, es posible construir una matriz simétrica M de tamaño $n \times n$ donde $M[i, j]$ almacena el máximo valor del flujo entre dos vértices distintos i y j.

La matriz M también puede ser representada empleando un árbol de expansión ponderado [77], donde cada arista representa a un corte fundamental de G y cuyo peso es equivalente al corte mínimo correspondiente. El máximo flujo $M[i, j]$ entre cualquier par de vértices i y j puede ser obtenido encontrando el único camino existente entre los vértices i y j en el árbol de expansión, y viene determinado por el mínimo peso sobre todas las aristas a lo largo del camino.

Por construcción se tiene que el máximo corte fundamental de G, es decir, el máximo flujo entre un par de vértices cualesquiera de G, es una cota inferior del CMP. Sea $T(V, E', w)$ el árbol de expansión ponderado de G calculado según [77], entonces se tiene:

$$CMP(G) \geq \max_{e \in E'} w(e)$$

2.1.5. Variantes del problema

Existen una serie de problemas que podrían ser considerados como variantes del CMP. Entre ellos destacan los siguientes:

- ***Modified Cutwidth***: variante del CMP consistente en que el cómputo del corte en cada posición excluye las aristas incidentes en el vértice que ocupa la posición evaluada. Esta variante es NP-Completa incluso para grafos planos con grado máximo 3 [129].

- ***Weighted Min Cut Problem***: variante consistente en considerar pesos en las aristas del grafo. La contribución de cada arista a la función objetivo, se ve afectada por el peso de la misma. Esta variante es NP-Completa incluso para árboles ponderados [129].

- ***Directed Cutwidth***: consiste en encontrar una ordenación topológica de los vértices de un grafo dirigido acíclico $G(V, A)$, de modo que para cada arco $a(u, v) \in A$ que une los vértices $u, v \in V$, la posición del vértice u en la ordenación sea menor que la posición del vértice v. Este problema es NP-Completo [52] en general, aunque puede ser resuelto polinómicamente para árboles [1].

- ***Cyclic Cutwidth***: el CMP puede ser considerado como un problema en el que se desea minimizar un determinado parámetro al embeber un grafo en una línea. Es posible considerar otro tipo de estructuras en las que embeber un grafo, tales como un ciclo [34]. En este caso, el valor del corte entre cada dos vértices consecutivos, viene dado por todas aquellas aristas que crucen la región triangular formada entre los vértices considerados y el centro del círculo que representa el ciclo.

2.1.6. Algoritmos polinómicos para grafos específicos

A pesar de que el CMP es NP-Difícil en general, existen algunas clases de grafos particulares, para las que se conocen algoritmos eficaces, capaces de obtener soluciones óptimas al problema en tiempo polinómico. En la Tabla 2.1 se recogen los grafos para los que existen algoritmos de este tipo, así como su correspondiente complejidad.

Tipo de grafo	Complejidad	Referencia
árboles	$O(n\, log^{\Delta-2} n)$	[35]
árboles	$O(n \log n)$	[186]
árboles binarios completos	$O(n)$	[105]
bipartite permutation	$O(n)$	[86]
bounded degree partial w-tree	$n^{O(wd)}$	[178]
c-ary cliques de d-dimensiones	$O(n)$	[136, 137]
grafos completos *p-partite*	$O(n + p\log(p))$	[133]
hipercubos	$O(n)$	[83, 138]
mallas de 2- y 3- dimensiones	$O(n^2)$	[159]
mallas toroidales y cilíndricas de 2-dimensiones	$O(n^2)$	[159]
mallas toroidales de 3-dimensiones	$O(n^2)$	[159]
máx grado $\leq \Delta$ y *treewidth* $\leq k$	$O(n^{\Delta k^2})$	[175]
threshold	$O(n)$	[85]
unit interval	$O(n)$	[187]

Tabla 2.1: Clases de grafos resolubles de manera óptima en tiempo polinómico, siendo n el número de vértices del grafo y Δ el grado máximo de un vértice del grafo.

2.1.7. Grafos con óptimo conocido

Pese a que no se conoce el valor óptimo al CMP para la gran mayoría de tipos grafos, existen algunos grafos particulares que poseen una estructura bien definida, para los cuales el valor óptimo sí que es conocido. Esto quiere decir que es posible determinar el valor del CMP basándose, por ejemplo, en la estructura del grafo o en el número de vértices del mismo, empleando para ello algún tipo de fórmula o expresión matemática. A continuación, se repasan algunos de estos grafos:

- **Rejilla (*Grid/Mesh*)**: tipo de malla bidimensional (*bidimensional mesh*) que podría ser definido como el producto cartesiano de dos caminos ($P_m \times P_n$) [149]. Un camino P_n en un grafo $G(V, E)$ es una secuencia de vértices $(v_1, v_2, \ldots, v_n)$ tal que, todo par de vértices consecutivos está unido por una arista de E. Puede verse un ejemplo de *grid* de tamaño 3x3 y otro de tamaño 5x7 en la Figura 2.2.

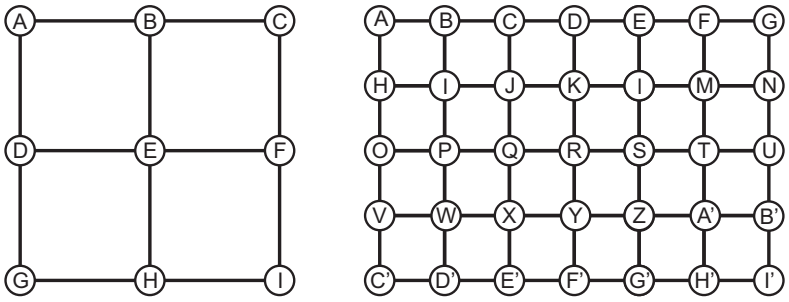


Figura 2.2: *Grid* de 9 vértices (*Grid* 3x3) y *grid* de 35 vértices (*Grid* 5x7).

El óptimo para este tipo de grafos para el problema del CMP fue formulado en [159] y se puede definir de la siguiente manera:

$$\forall\, m, n \geq 2,\ \mathrm{CMP}(P_m \times P_n) = \begin{cases} 2, & \text{si } m = n = 2 \\ \min\{m+1, n+1\}, & \text{en cualquier otro caso} \end{cases}$$

A partir de la definición anterior, los *grid* de tamaño 3x3 y 5x7, de la Figura 2.2 tendrían un valor óptimo de 4 y 6 respectivamente.

Además de conocer el valor óptimo del CMP dada la estructura de un *grid* determinado, es posible determinar cómo etiquetar sus vértices de manera óptima. En concreto están documentados dos etiquetados óptimos para este tipo de grafo: el etiquetado lexicográfico y el etiquetado diagonal. Puede verse un ejemplo de estos etiquetados en la Figura 2.3.

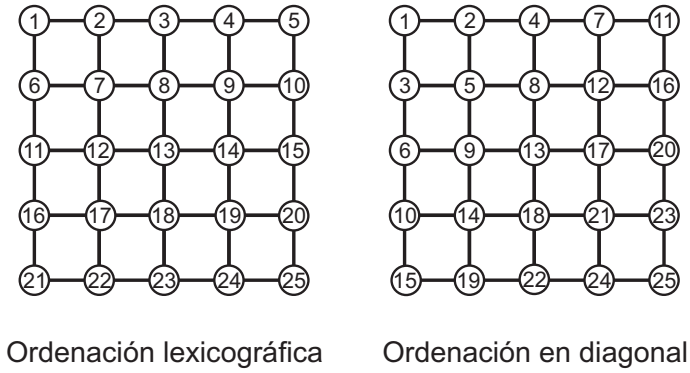


Figura 2.3: Etiquetados óptimos para un *Grid* 5x5.

- **Otras mallas bidimensionales**: además de los *grid*, existen otro tipo de mallas bidimensionales para las que también se conoce el valor óptimo para el CMP como son, por ejemplo, las mallas definidas como el producto cartesiano de un camino y un ciclo ($P_m \times C_n$) y las mallas definidas como el producto cartesiano de dos ciclos ($C_m \times C_n$). Un ciclo C_n en un grafo $G(V, E)$ es un camino en el que el vértice de partida y llegada coinciden. La fórmula para calcular el valor óptimo de estos grafos para el CMP fue definida en [159] y puede ser resumida de la siguiente manera:

$$\forall\, m, n \geq 3,\ \mathrm{CMP}(C_m \times C_n) = \min\,\{2m+2, 2n+2\}$$

$$\forall\, m \geq 2, n \geq 3,\ \mathrm{CMP}(P_m \times C_n) = \begin{cases} 4, & \text{si } m = 2, n = 3 \\ \min\,\{2m+1, n+2\}, & \text{en cualquier otro caso} \end{cases}$$

- **Árboles completos de grado m (m-*ary trees*)**: los árboles son casos especiales de grafos conexos acíclicos, en los que cada par de vértices está conectado exactamente por un camino. Un camino es una sucesión de vértices en la que cada uno de sus vértices tiene una arista hacia el vértice sucesor. Dos nodos conectados por una arista permiten establecer las relaciones de «padre» e «hijo». Todo nodo (excepto la raíz) tiene un nodo padre y puede tener cero o varios nodos hijo. Los nodos sin hijos se denominan nodos hoja. Los nodos que no son nodos hoja se denominan nodos internos. La altura de un árbol es el número de vértices del mayor camino entre la raíz y un nodo hoja.

  Un árbol completo de grado m es aquél en el que todos los nodos hoja están al mismo nivel y todos los nodos internos tienen exactamente m hijos. Puede verse un ejemplo de árbol completo de grado 2 y altura 4 en la Figura 2.4.

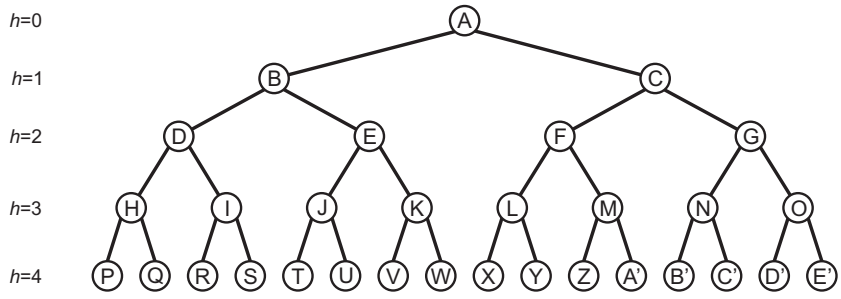


Figura 2.4: Árbol completo de grado 2 y altura 4.

Siendo T_h^m el árbol completo de grado m y altura h (considerando que la raíz del árbol se encuentra en el nivel 0) se puede calcular su $\mathrm{CMP}(T_h^m)$, mediante la expresión [105]:

$$\mathrm{CMP}(T_h^m) = \begin{cases} 0, & \text{si } h = 0 \\ \lceil \frac{m}{2} \rceil, & \text{si } h = 1 \\ m, & \text{si } h = 2 \\ \mathrm{CMP}^*(T_{h-1}^m) + \lceil \frac{m}{2} \rceil - 1 & \text{si } h \geq 3 \end{cases}$$

$$\mathrm{CMP}^*(T_h^m) = \begin{cases} 1, & \text{si } h = 0 \\ \lfloor \frac{m}{2} \rfloor + 1, & \text{si } h = 1 \\ 2\lfloor \frac{m}{2} \rfloor + 1, & \text{si } h = 2 \\ \mathrm{CMP}^*(T_{h-2}^m) + m - 1 & \text{si } h \geq 3 \end{cases}$$

- **Grafos completos**: un grafo completo es aquél en el que cada par de vértices del grafo está unido por una arista. Un grafo completo con n vértices (K_n) tiene $\binom{n}{2} = n(n-1)/2$ aristas. En la Figura 2.5 se muestran ejemplos de grafos completos de 5 y 6 vértices. Los grafos completos son un caso trivial para el CMP, ya que con independencia de cómo se etiqueten los vértices del mismo, el valor de función objetivo no varía.

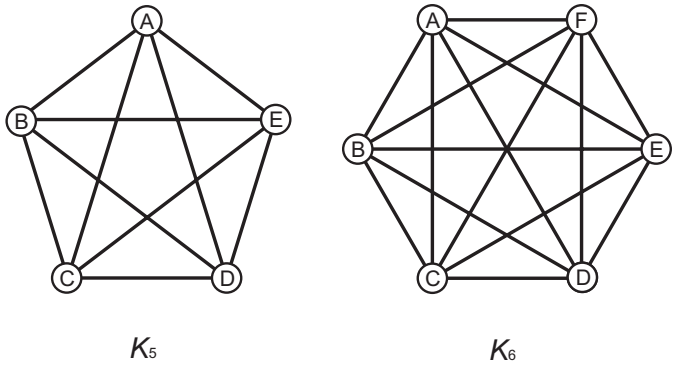


Figura 2.5: Ejemplo de grafos completos de 5 vértices (K_5) y 6 vértices (K_6).

El valor del CMP para un grafo de n vértices se puede calcular empleando la siguiente fórmula [158]:

$$\text{CMP}(K_n) = \begin{cases} \frac{n^2}{4}, & \text{si } n \text{ es par} \\ \\ \frac{n^2-1}{4}, & \text{si } n \text{ es impar} \end{cases}$$

Conocer el óptimo de un grafo dado es de especial interés a la hora de evaluar la calidad de los algoritmos propuestos. En concreto, permite calcular la desviación de las soluciones aportadas por un algoritmo al valor óptimo para ese grafo.

2.2. Publicaciones relacionadas con el CMP

Las publicaciones relacionadas con el CMP son numerosas, aunque no todas ellas tienen relación directa con el enfoque de esta Tesis Doctoral. A pesar de ello, es de especial interés establecer una panorámica de las mismas que ayude a una mejor comprensión de la relevancia del problema abordado, así como de la propuesta realizada. Aplicaciones prácticas; algoritmos polinómicos para grafos específicos; algoritmos para la versión parametrizada del problema; algoritmos heurísticos; formulación entera o estudio de la complejidad del problema, son algunas de las temáticas abordadas en las publicaciones disponibles en el estado del arte.

En la Figura 2.6 se han representado, a modo de histograma, las publicaciones más relevantes relacionadas con el CMP ordenadas cronológicamente. Como se puede observar, las primeras publicaciones relacionadas con el CMP se remontan a la década de los sesenta, si bien no es

hasta los años noventa cuando aparece un mayor número de publicaciones. Además del orden cronológico, se ha realizado también una clasificación temática de las mismas, empleando para ello diferentes colores. Es importante destacar que algunos de los trabajos recogidos podrían ser clasificados en más de una categoría aunque, por simplicidad, se ha decidido incluirlos sólo en una. En concreto, las publicaciones están agrupadas en las siguientes categorías temáticas:

- **Algoritmos aproximados**: categoría que incluye aquellos trabajos en los que se proponen algoritmos aproximados para el CMP, que ofrecen una garantía teórica de la proximidad al óptimo.

- **Aplicaciones**: esta categoría incluye aquellos trabajos en los que el CMP es utilizado de manera práctica en otras disciplinas, o como medio para resolver otros problemas.

- **Complejidad**: categoría que recoge las publicaciones en las que se estudia la complejidad computacional que entraña la resolución del CMP. Existen trabajos que estudian dicha complejidad para grafos genéricos y otros que lo hacen para alguna familia de grafos específica.

- **Formulación entera**: esta categoría está dedicada a las formulaciones enteras del CMP presentes en el estado del arte. Aunque la categoría está formada únicamente por un trabajo, se ha considerado relevante destacar dicha publicación por la relación directa que tiene con esta Tesis Doctoral.

- **Grafos específicos**: categoría que aglutina las publicaciones en las cuales se proponen algoritmos polinómicos para algún tipo de grafo particular o para alguna familia de grafos.

- **Heurísticos**: categoría que recoge los trabajos en los que se aborda el CMP empleando un enfoque heurístico.

- **Versión parametrizada**: bajo esta categoría se agrupan los trabajos en los que se estudia la versión parametrizada del problema. El objetivo es estudiar qué hace difícil al problema, dividiendo la entrada en dos partes, una parte no parametrizada y otra parametrizada en la que se imponen ciertas restricciones.

- **Otros trabajos relacionados**: en esta categoría se engloban otros trabajos que guardan algún tipo de relación con el CMP. En concreto se trata de trabajos en los que se abordan aspectos tales como: variantes del problema, cotas al problema, otros problemas de ordenación lineal relacionados, etc.

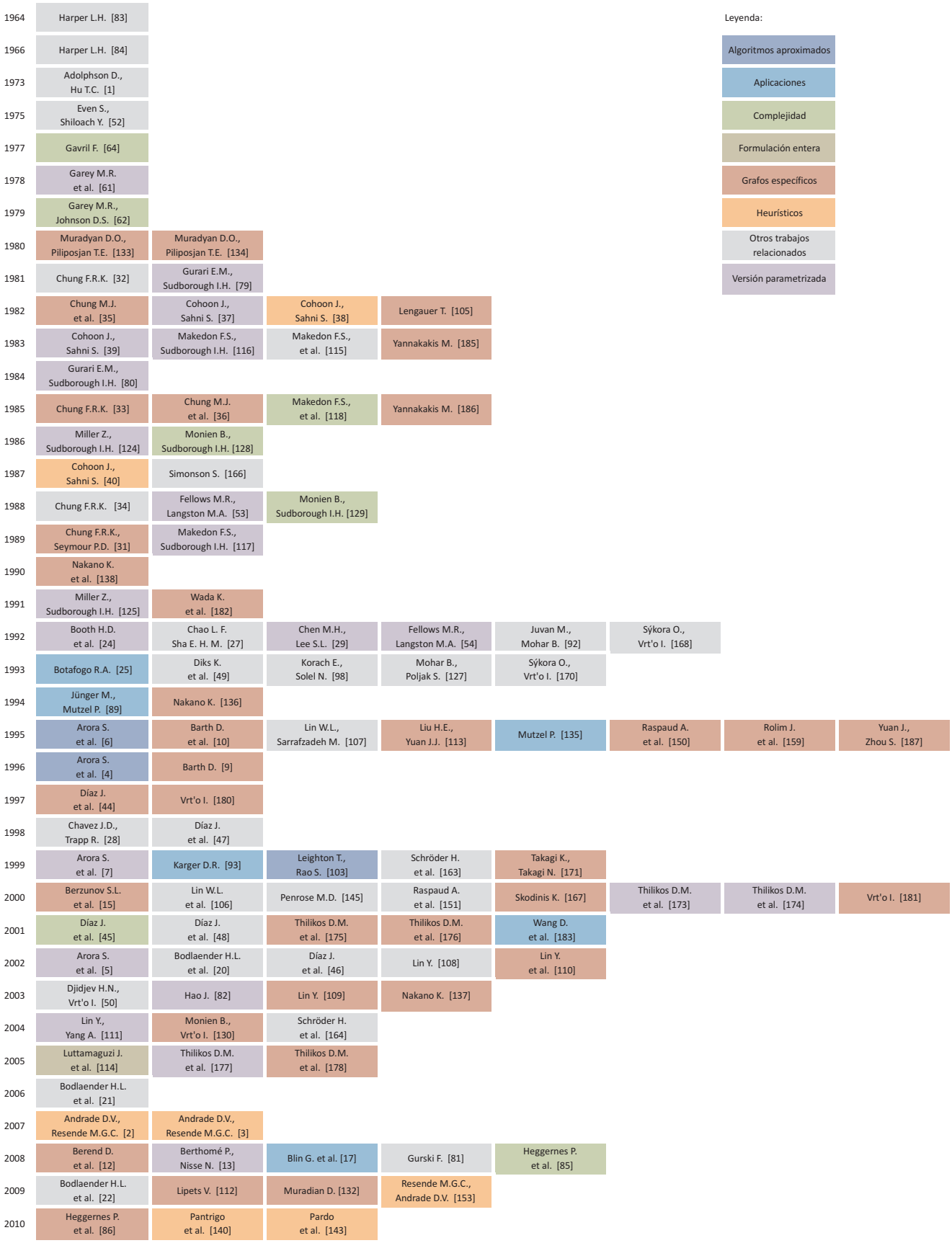


Figura 2.6: Histograma cronológico-temático de publicaciones relacionadas con el CMP.

Como se puede observar en la Figura 2.6 existe un gran número de trabajos en los que se aborda el CMP para algún tipo de grafo específico, así como trabajos en los que se estudia la versión parametrizada del mismo. De entre los trabajos recogidos, son de especial relevancia los trabajos en los que se estudia la complejidad computacional del problema [45, 62, 64, 128, 129].

En cuanto al enfoque de esta Tesis Doctoral se refiere, destaca el bajo número de algoritmos heurísticos propuestos. En concreto y pese a que aparecen divididos en diversas publicaciones ([3, 2, 38, 40, 153]) sólo se han encontrado dos propuestas diferentes. Estas publicaciones están denotadas en color naranja en el histograma de la Figura 2.6. Destaca también la ausencia de algoritmos exactos para grafos genéricos, si bien se ha encontrado una formulación entera para el CMP [114] denotada en color beis oscuro en el histograma.

Una vez definidas las distintas temáticas que se han abordado en relación con el CMP, puede ser de interés estudiar también el tipo de publicaciones a que han dado lugar los distintos trabajos. Esto, ayuda a dar una idea del interés que tiene el problema. En la Figura 2.7 se recoge, también a modo de histograma cronológico, dónde han sido publicados los trabajos relacionados con el CMP. En concreto se han considerado las siguientes categorías:

- **Congresos**: incluye las publicaciones en congresos de ámbito nacional o internacional relacionadas con el CMP.
- **Capítulos de libro**: recoge los trabajos relacionados con problema que han sido publicados como capítulos de libro o *Lecture Notes in Computer Science* [4] (LNCS).
- **Otras revistas**: incluye todas aquellas publicaciones en revistas no indexadas en el *Journal Citation Reports* (JCR)[5].
- **Patentes**: aunque esta categoría sólo contiene una publicación, se ha considerado importante destacarla por su especial relación con esta Tesis Doctoral.
- **Revistas indexadas en el JCR**: categoría que aglutina las publicaciones recientes de mayor impacto relacionadas con el CMP.
- **Informes técnicos**: recoge versiones preliminares de otros trabajos o estudios llevados a cabo que no han sido publicados.

Como se puede observar en la Figura 2.7 en los últimos años han proliferado las publicaciones de impacto relacionadas con el CMP. Destacar también la reciente publicación de una patente [153]. Por otro lado, también son numerosas las publicaciones presentadas en congresos y revistas no indexadas en el JCR, siendo un poco menor el número de capítulos de libro publicados.

[4] `http://www.springer.com/computer/lncs`
[5] No se dispone de datos anteriores a 1997 para este índice de impacto

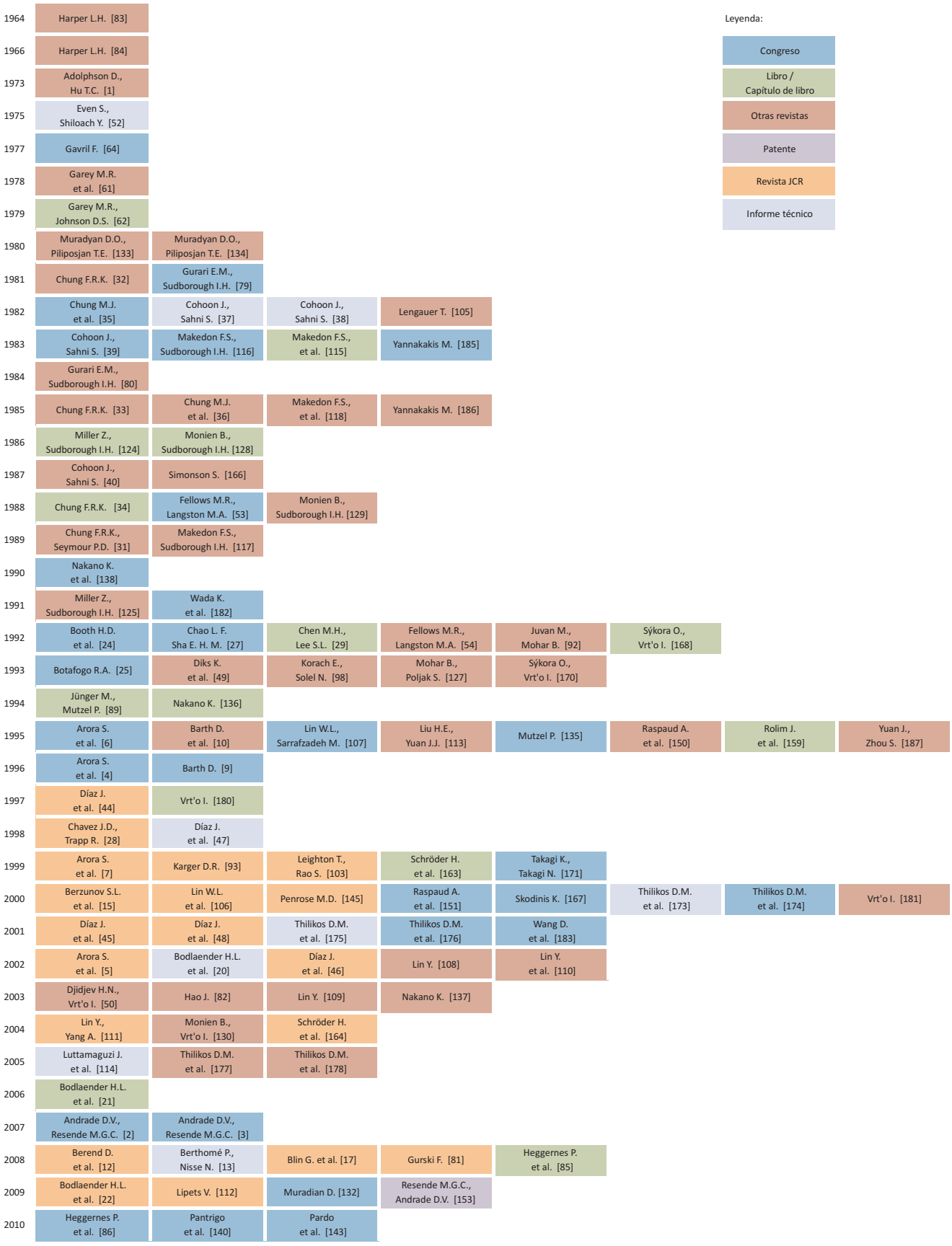


Figura 2.7: Histograma cronológico que recoge dónde han sido publicados los trabajos relacionados con el CMP.

2.3. Revisión de propuestas relacionadas con un enfoque exacto

Partiendo del enfoque propuesto en esta Tesis Doctoral para la resolución exacta del CMP, no se han encontrado en la literatura algoritmos exactos que aborden el problema para grafos genéricos. No obstante, existe un trabajo, titulado «*Integer programming solutions for several optimization problems in graph theory*» [114] en el que se propone una formulación entera para el problema.

Los problemas de optimización pueden ser expresados como problemas de Programación Entera (IP, del inglés *Integer Programming*). A partir de una formulación IP para un problema de optimización, se pueden utilizar *solver* de uso general para encontrar soluciones óptimas o, en su defecto, acotar la solución a instancias del problema. Los *solver* se podrían definir como programas que permiten resolver problemas de carácter general. Para ello hacen uso de técnicas basadas en Planos de Corte y Ramificación y Poda, si bien necesitan la formulación entera del problema, que debe ser facilitada.

Disponer de una formulación para el CMP, permite medir la calidad de los resultados obtenidos por los algoritmos de Ramificación y Acotación propuestos, al compararlos con los resultados obtenidos por un *solver* empleando una formulación IP. A continuación se repasa la formulación IP para el CMP propuesta en [114].

2.3.1. Representación en Programación Entera del CMP

Dado un grafo $G(V, E)$ con $|V| = n$, se pretende encontrar una ordenación f de los vértices de G tal que $f : V \longrightarrow \{1, 2, \ldots, |V|\}$. Dada la definición del problema, para describir la formulación IP del mismo, los autores introducen la definición de permutación en IP y de la operación lógica conjunción. Ambas definiciones son necesarias para representar el CMP en IP y se repasan a continuación.

Representación de una permutación en IP

Sean X_i^k las variables de decisión binarias (con $i, k \in [0, n-1]$) que especifican el hecho de que el vértice i se asigne a la posición k de la permutación. Es decir, $\forall i, k \in [0, n-1], X_i^k \in \{0, 1\}$, luego X_i^k será igual a 1 si y sólo si el vértice i está en la posición k.

Para caracterizar la correspondencia 1-1 de tal asignación, se añaden las siguientes restricciones:

- $\forall\, i \in [0, n-1]$, $\sum_{k\in[0,n-1]} X_i^k = 1$. Esta expresión indica que cada vértice tiene asignado exactamente una posición en la ordenación lineal.

- $\forall\, k \in [0, n-1], \sum_{i \in [0,n-1]} X_i^k = 1$. Esta expresión indica que cada posición está asignada exactamente a un vértice en la ordenación lineal.

Por lo tanto, las restricciones de una permutación en IP, dadas las variables de decisión binarias X_i^k, se podrían sintetizar en:

- $X_i^k \in \{0, 1\}$ con $i, k \in [0, n-1]$
- $\forall i \in [0, n-1]$, $\sum_{k \in [0,n-1]} X_i^k = 1$
- $\forall k \in [0, n-1]$, $\sum_{i \in [0,n-1]} X_i^k = 1$

Operación lógica conjunción

Antes de pasar a definir el CMP en IP, es necesario introducir la operación conjunción de dos variables binarias, denotada mediante el operador $\wedge$, y que aparece descrita en [114]. Sean x, y, z variables binarias tales que $z = x \wedge y$, entonces se deben cumplir las siguientes restricciones:

$$z = x \wedge y \mapsto \begin{cases} z \leq x \\ z \leq y \\ x + y \leq z + 1 \end{cases}$$

Para constatar el hecho de que el vértice i esté en la posición k, o no lo esté, es necesario utilizar la variable binaria X_i^k. El resultado de la conjunción de dos variables binarias X_i^k y X_j^l, quedaría representado por la variable binaria $Y_{i,j}^{k,l}$:

$$X_i^k \wedge X_j^l = Y_{i,j}^{k,l}$$

El sentido de esta operación, permite determinar si, para una ordenación dada, los vértices i, j están asignados a las posición k, l respectivamente, de manera simultánea.

Una posible formulación del CMP en IP

Como se ha mencionado anteriormente, en [114] se propone la única formulación IP para el CMP que se ha encontrado propuesta en trabajos previos. Dado un grafo $G(V, E)$ el CMP consiste en encontrar una ordenación lineal f de los vértices de G de manera que el $CW_f(G)$ sea minimizado.

- Ordenación: $f : V \longrightarrow \{1, 2, \ldots, |V|\}$
- $CW_f(G)$: $\max_{1 \leq c \leq |v|-1} \mid \{(v_i, v_j) \in E : f(i) \leq c < f(j)\} \mid$

donde $c \in [1, |V| - 1]$ representa cada uno de los vértices para los que es necesario evaluar el valor del *cutwidth*. Definidas las variables de decisión X_i^k e $Y_{i,j}^{k,l}$ junto con las restricciones de

una permutación y de una conjunción en programación entera, se define el CMP tal y como sigue:

Minimizar S,

sujeto a:

$$X_i^k \in \{0,1\},\ \forall\, i,k \in \{1,\ldots,n\} \tag{2.1}$$

$$\textstyle\sum_{k\in\{1,\ldots,n\}} X_i^k = 1,\ \forall\, i \in \{1,\ldots,n\} \tag{2.2}$$

$$\textstyle\sum_{i\in\{1,\ldots,n\}} X_i^k = 1,\ \forall\, k \in \{1,\ldots,n\} \tag{2.3}$$

$$Y_{i,j}^{k,l} \leq X_i^k \tag{2.4}$$

$$Y_{i,j}^{k,l} \leq X_j^l \tag{2.5}$$

$$X_i^k + X_j^l \leq Y_{i,j}^{k,l} + 1 \tag{2.6}$$

$$\textstyle\sum_{(k\leq c<l)\vee(l\leq c<k)} Y_{i,j}^{k,l} \leq S,\ \forall\, c \in \{1,\ldots,n-1\} \tag{2.7}$$

donde X_i^k son variables de decisión binarias con índices $i,k \in \{1,2,\ldots,n\}$ que especifican si el vértice i está asignado a la posición k en la ordenación. Las restricciones (2.2) y (2.3) aseguran que cada vértice está asignado únicamente a una posición y cada posición únicamente a un vértice, respectivamente. Por lo tanto, las restricciones (2.1), (2.2) y (2.3) juntas, especifican que la solución al problema es una ordenación. La variable de decisión $Y_{i,j}^{k,l} \in \{0,1\}$ definida en términos de X_i^k y X_j^l determina que el vértice i esté en la posición k y que el vértice j esté en la posición l al mismo tiempo (restricciones (2.4), (2.5) y (2.6)). Por último, la restricción (2.7) representa la evaluación, para cada vértice c, del número de aristas que unen un vértice en cualquier posición k, $(1 \leq k < c)$ y otro vértice con una posición l, $(c < l \leq n)$.

2.4. Revisión de propuestas relacionadas con un enfoque heurístico

La resolución heurística del CMP para grafos generales constituye otra de las piedras angulares de esta Tesis Doctoral. Pese a que los algoritmos heurísticos se han mostrado eficaces a la hora de obtener soluciones de alta calidad a problemas complejos de optimización combinatoria, los investigadores en el área no han prestado demasiada atención al problema del CMP. En concreto, se pueden encontrar en el estado del arte las siguientes publicaciones relacionadas:

- «*Heuristics for the Board Permutation Problem*» [38]
- «*Heuristics for Backplane Ordering*» [40]
- «GRASP *with Path-Relinking for Network Migration Scheduling*» [3]
- «GRASP *with Evolutionary Path-Relinking*» [2]

- «*Method and System for Network Migration Scheduling*» [153]

Antes de describir en detalle cada una, conviene destacar que pese a que se trate de cinco publicaciones diferentes, en realidad son sólo dos propuestas distintas. Una propuesta agrupa a las dos primeras publicaciones, donde [38] es un informe técnico que posteriormente dio lugar a una versión publicada en una revista [40]. Por otro lado, las tres últimas publicaciones [3, 2, 153] (dos publicadas en congresos internacionales [3, 2] y una como patente [153]) son diferentes versiones del mismo algoritmo.

A continuación, en las secciones 2.4.1 y 2.4.2, se describirán en detalle las dos propuestas anteriormente mencionadas.

2.4.1. *Heuristics for Backplane Ordering*

En este trabajo [40] se aborda una generalización del problema del CMP llamada «*Backplane Ordering*» (aunque los mismos autores también se han referido a ella como «*Board Permutation Problem*» [38]). Esta generalización tiene aplicación en el diseño de circuitos integrados.

Cuando se diseñan sistemas digitales complejos, a menudo son divididos en unidades funcionales, cada una de las cuales es diseñada e implementada por separado. El resultado es una colección de unidades independientes o dispositivos que, al ser conectados, forman el sistema digital deseado. Conociendo qué dispositivos tienen que ser conectados entre sí, se puede establecer un orden para los mismos antes de conectarlos ya que, en función de dicha ordenación, se mejorarán aspectos claves en el diseño del circuito, tales como el número de canales necesarios o el espacio que se necesitará para albergar ese sistema. Los dispositivos, junto con sus interconexiones son conocidos como el «*backplane*» del circuito.

El objetivo de este problema consiste es encontrar una ordenación de los dispositivos que minimice el área destinada al *backplane*. Una técnica para minimizar este área consiste en minimizar el máximo número de interconexiones entre los dispositivos. Esta técnica se basa en la idea de que cada interconexión requiere espacio y que, por lo tanto, minimizar el número de interconexiones minimiza también el área necesaria [1, 162, 170].

El problema anteriormente descrito puede ser modelado como un hipergrafo $H(B, L)$ donde B es un conjunto $\{b_1, b_2, \ldots, b_n\}$ de dispositivos con $|B| = n$ (que se corresponden con los vértices de H) y L es un conjunto de redes $\{N_1, N_2, \ldots, N_m\}$ de B tal que $|L| = m$ (que representan las hiperaristas de H). Una hiperarista puede unir cualquier número de vértices, o lo que es lo mismo en este caso, cualquier número de dispositivos. El CMP supone un caso particular de este problema, en el que todas y cada una de las redes N_i están formadas únicamente por dos dispositivos ($|N_i| = 2$); es decir, cada hiperarista conecta únicamente dos vértices. Esta restricción convierte el hipergrafo en un grafo.

En este trabajo se proponen diversos algoritmos constructivos y un conjunto de métodos de mejora para abordar el problema del «*backplane ordering*». Los autores emplean la técnica del Enfriamiento Simulado (SA, del inglés *Simulated Annealing* [96]) para combinar su mejor estrategia constructiva con su mejor método de mejora.

A continuación, se describen en detalle los métodos constructivos y de búsqueda local más destacados, así como la combinación más efectiva de los anteriores.

Heurísticas constructivas

Se proponen tres heurísticas constructivas diferentes (denominadas «H1», «H2» y «H3») que posteriormente serán combinadas con distintas estrategias de búsqueda local.

Cada heurística constructiva parte de una solución vacía y va añadiendo vértices a la solución, de uno en uno, hasta que todos los vértices del grafo sean etiquetados. A cada paso, todos los vértices no etiquetados son candidatos a ser el siguiente vértice a etiquetar. El criterio para seleccionar el siguiente vértice a etiquetar consiste en elegir aquel vértice que introduzca menos redes nuevas (aristas) a la solución parcialmente construida. Es decir, aquel vértice que minimice el número de aristas entre el conjunto de vértices etiquetados y el conjunto de vértices no etiquetados.

La diferencia entre las distintas heurísticas constructivas propuestas reside en la posición que ocupará el siguiente vértice a etiquetar. Dichas estrategias son repasadas a continuación:

- H1: en cada iteración, el nuevo vértice seleccionado recibirá la etiqueta más baja aún disponible. Es decir, se asignará inicialmente la etiqueta 1, después la 2 y así sucesivamente hasta asignar la etiqueta n.
- H2: en cada iteración se asignan alternativamente las etiquetas 1, n, 2, n-1, etc. Es decir, se etiqueta el grafo desde los extremos hacia el centro.
- H3: en cada iteración del algoritmo, se asignará la etiqueta (i) más baja disponible. La diferencia con la «H1» reside en que el nuevo vértice a etiquetar no recibirá necesariamente la etiqueta i, sino aquella etiqueta comprendida entre 1 e i que minimice el valor de la función objetivo.

De entre las diferentes heurísticas constructivas propuestas, los autores destacan «H1» como la que produce soluciones de mejor calidad. El pseudocódigo de esta heurística se muestra en el Algoritmo 2.1. En él, inicialmente se parte de una solución vacía (paso 2) y todos los vértices del grafo pertenecen al conjunto de vértices no etiquetados (NE) siendo, por lo tanto, candidatos a ser etiquetados en la siguiente iteración del algoritmo. Por otro lado, el conjunto de vértices etiquetados (E) está inicialmente vacío. Entre los pasos 7 y 13 del algoritmo, se evalúa la hipotética aportación de cada vértice no etiquetado ($b \in NE$) a la función objetivo,

si éste fuera asignado a la posición i. De entre todos los vértices evaluados se selecciona el que menos incremente la función objetivo (b^*) y se le asigna la posición i de la solución (paso 14). Los vértices escogidos en cada iteración son etiquetados en orden lexicográfico, es decir, en la primera iteración $i = 1$ y en la última $i = n$, siendo n el número de vértices del grafo.

Algoritmo 2.1 Algoritmo constructivo «H1» (J.P. Cohoon and S. Sahni [40]).

```
1:  procedimiento H1()
2:  π ← ∅ ;
3:  NE ← B ;
4:  E ← ∅ ;
5:  para i ← 1 hasta n hacer
6:      min ← ∞
7:      para todo b ∈ NE hacer
8:          K ← evaluar-aristas(π, NE, E, b);
9:          si K < min entonces
10:             min ← K ;
11:             b* ← b ;
12:         fin si;
13:     fin para;
14:     añadir(π, i, b*);
15:     NE ← NE − {b*};
16:     E ← E + {b*};
17: fin para
18: devolver (π);
19: fin H1;
```

Búsqueda local

Los métodos propuestos para la mejora de las soluciones están basados en dos estrategias diferentes: intercambios e inserciones. Para ambas estrategias se ha empleado un enfoque exhaustivo que consiste en probar, uno a uno, todos los posibles movimientos a partir de una solución dada, y mantener aquellos movimientos que conduzcan a una solución de mejor calidad, siguiendo un enfoque *first improvement*.

La búsqueda local basada en intercambios, denominada «H4», examina la solución de entrada para determinar si es posible realizar algún intercambio de vértices que reduzca el valor de la función objetivo. En el caso de que algún intercambio mejore la solución en curso, éste se realiza y el proceso se repite. La heurística finaliza cuando, tras examinar todos los posibles intercambios de una solución dada, no es posible realizar ningún intercambio de mejora.

Por otro lado, la búsqueda local basada en inserciones, denominada «H9», tiene un comportamiento similar a «H4», sustituyendo intercambios por inserciones. Este método examina, uno a uno, todos los posibles movimientos de inserción que se pueden llevar a cabo con cada uno de los vértices de la solución. Si se encuentra un movimiento que mejore el valor de la función objetivo, éste se mantiene y el proceso se repite. En cambio, si ninguna inserción

consigue mejorar la solución, el proceso finaliza.

Simulated Annealing

Una vez planteadas las dos estrategias exhaustivas de búsqueda local anteriormente mencionadas («H4» y «H9») los autores proponen embeber dichas estrategias en un esquema SA.

El método SA está basado en una técnica previa desarrollada por Metropolis et al. [123] en el campo de la termodinámica estadística. El algoritmo de Metropolis simula el comportamiento de un sistema térmico a alta temperatura durante el proceso de enfriamiento del mismo. Este comportamiento viene determinado por el desplazamiento de partículas del sistema, a medida que decrece la temperatura, hasta que el sistema converge a un estado estable. La simulación se realiza empleando variaciones aleatorias y un modelo probabilístico, a través del cual se pretende decidir si aceptar o no un determinado movimiento aleatorio, que representa un cambio térmico de energía.

En la década de los ochenta, Kirkpatrick et al. [96] mostraron cómo este proceso podía ser aplicado a problemas de optimización. Cualquier implementación de búsqueda local puede convertirse en una implementación de SA, eligiendo elementos del entorno de modo aleatorio, aceptando todos los movimientos que mejoren la solución, y aceptando algunos movimientos que la empeoren basándose en la probabilidad determinada por una función. Es interesante destacar que Cerny [26], de manera independiente, llegó a las mismas conclusiones trabajando en el contexto del Problema del Viajante (TSP, del inglés *Traveling Salesman Problem*).

Los autores de este trabajo propusieron variantes de «H4» y de «H9» basadas en la técnica de SA. Las nuevas heurísticas fueron denominadas «H5» y «H10» respectivamente. El pseudocódigo de «H10» (que a la postre formará parte del mejor algoritmo propuesto en este trabajo) se muestra en el Algoritmo 2.2. En él se puede observar como el procedimiento comienza recibiendo una solución ya formada (τ) que se almacena como la mejor solución encontrada hasta la fecha (paso 2). A continuación se evalúa la solución (paso 4) y se procede a realizar inserciones con cada uno de los vértices (i) de la solución, en cada una de las posiciones (j) (pasos 5 a 18). Durante este proceso, cada vez que se realiza una inserción, generando así una nueva solución (paso 14) se evalúa el coste asociado a la misma (paso 15). Si la nueva solución es mejor que la mejor solución encontrada se reinicia la búsqueda comenzando otra vez con el primer vértice. En cambio, si tras realizar todas las posibles inserciones de todos los vértices en todas las posiciones posibles, no se ha producido ningún movimiento de mejora, se determina con una cierta probabilidad si se sigue la exploración o se finaliza el proceso (paso 22). En el caso de finalizar la exploración, se devuelve la mejor solución encontrada (paso 23), en caso contrario, se realiza un movimiento aleatorio en la solución (paso 25) y se repite el proceso de búsqueda.

Algoritmo 2.2 Búsqueda local basada en SA, «H10» (J.P. Cohoon and S. Sahni [40]).

```
1: procedimiento H10(τ)
2: π ← τ ;
3: mientras true hacer
4:       coste ← evaluar(τ);
5:       repetir
6:             ρ ← τ;
7:             min ← coste ;
8:             i ← 0;
9:             repetir
10:                  i ← i + 1;
11:                  j ← 0;
12:                  repetir
13:                        j ← j + 1;
14:                        τ ← insertar(ρ, i, j);
15:                        coste ← evaluar(τ);
16:                  hasta que (j = n) o (coste < min);
17:            hasta que (i = n) o (coste < min);
18:      hasta que coste ≥ min;
19:      si evaluar(ρ) < evaluar(π) entonces
20:            π ← ρ ;
21:      fin si;
22:      si rechazar() entonces
23:            devolver (π);
24:      si no
25:            Realizar un movimiento aleatorio en ρ;
26:            τ ← ρ ;
27:      fin si;
28: fin mientras
29: fin H10;
```

Combinación de métodos constructivos y de mejora

Finalmente, los autores de este trabajo, proponen la combinación de los tres algoritmos constructivos descritos anteriormente («H1», «H2» y «H3») con los métodos de mejora basados en la técnica de *Simulated Annealing* («H5» y «H10»). El resultado, son un conjunto del algoritmos de dos fases en los que, en una primera fase se construye una solución y, en una segunda fase, se mejora. Los algoritmos resultantes son los siguientes:

- «H6»: combinación de «H1» y «H5»
- «H7»: combinación de «H2» y «H5»
- «H8»: combinación de «H3» y «H5»
- «H11»: combinación de «H1» y «H10»
- «H12»: combinación de «H2» y «H10»

- «H13»: combinación de «H3» y «H10»

Como resultado de estas combinaciones, los autores determinan que la heurística más eficiente de todas las propuestas en [40] es «H11», formada por la combinación de «H1» y de «H10».

2.4.2. GRASP *with Path-Relinking for Network Migration Scheduling*

Este trabajo ha dado lugar a diversas publicaciones [3, 2, 153] que suponen diferentes evoluciones de la misma propuesta. En él, se aborda el CMP con el objetivo de resolver una aplicación práctica del problema (*Network Migration Scheduling*) que consistente en migrar una serie de dispositivos conectados entre sí en una red de telecomunicaciones, a otra red más moderna, manteniendo la misma estructura. En cada paso del proceso de migración, todas las conexiones que parten de un nodo de la red antigua pasan a un nodo de la red nueva. El objetivo es minimizar el número de conexiones simultáneas entre la red antigua y la nueva.

El problema anteriormente descrito puede ser modelado como el CMP, representando la red de telecomunicaciones como un grafo y estableciendo un orden para migrar los nodos de la red (ordenación lineal de los vértices del grafo). El número de conexiones simultáneas entre la red antigua y la nueva viene determinado por el valor del *cutwidth* en cada posición de la ordenación lineal generada. Para abordar el problema anterior, los autores proponen un algoritmo tipo GRASP combinado con la estrategia de Reencadenamiento de Trayectorias (PR, del inglés *Path Relinking*). Poco después, amplían la propuesta añadiendo la técnica del Reencadenamiento de Trayectorias Evolutivo (EvoPR, del inglés *Evolutionary Path Relinking*).

A continuación, se repasa detalladamente el algoritmo propuesto, abordando por separado los siguientes aspectos del mismo: algoritmo constructivo, algoritmo de búsqueda local, GRASP con Reencadenamiento de Trayectorias y Reencadenamiento de Trayectorias Evolutivo.

Algoritmo constructivo

Los autores proponen un algoritmo constructivo tipo GRASP [55, 56, 156] en el que lleva a cabo una generalización de una idea publicada previamente [146]. La idea original parte de una solución inicial vacía y va añadiendo vértices a la solución uno a uno, de modo que cada nuevo vértice añadido se prueba en la primera y en la última posición de la solución parcialmente construida, asignando al vértice la posición que menos incremente la función objetivo. La generalización consiste en que cada vértice que se añada a la solución se pruebe en cada una de las posiciones disponibles de la solución parcialmente construida, en lugar de únicamente en la primera y última posición.

Para determinar el vértice a insertar en cada momento, el algoritmo parte de una ordenación inicial de los vértices del grafo, a partir de la cual se seleccionan los vértices en el orden

establecido. En [146] se proponían diferentes estrategias posibles para generar dicha ordenación inicial, entre las que se encontraban: ordenación aleatoria, recorrido en profundidad y recorrido en anchura. Además se proponía una versión aleatorizada de las dos dos últimas, que consiste en seleccionar de manera aleatoria el vértice inicial. De todas las propuestas iniciales se utilizó la versión aleatorizada del algoritmo recorrido en profundidad.

Búsqueda local

El algoritmo de búsqueda local propuesto está basado en movimientos de intercambio de vértices. Este movimiento consiste en seleccionar dos vértices de una ordenación e intercambiar sus posiciones. La ordenación de partida para la búsqueda local será aquélla proporcionada por el algoritmo constructivo descrito anteriormente. En concreto, en este algoritmo se propone examinar todos los posibles intercambios de la ordenación en busca de una mejora de la función objetivo. El intercambio a realizar será el primero que mejore el valor de la función objetivo (estrategia conocida como *first improvement*). La búsqueda local finaliza cuando ningún posible intercambio se convierte en un movimiento de mejora.

El pseudocódigo de esta búsqueda local puede verse en el Algoritmo 2.3 donde, inicialmente, el procedimiento recibe una solución π (paso 1). A continuación, mientras que haya mejoras en la solución (pasos 3 a 12) repite el proceso de búsqueda, intercambiando todos los posibles pares de vértices de la solución (pasos 5 a 11). El proceso finaliza cuando tras examinar todos los posibles intercambios, ninguno ha resultado como una mejora de la solución.

Algoritmo 2.3 Búsqueda local (D.V. Andrade y M.G.C. Resende [3]).

1: **procedimiento** Busqueda-local($\pi, \pi^{-1}, w, K, K^*, nK^*$)
2: $mejora \leftarrow$ cierto;
3: **mientras** $mejora$ = cierto **hacer**
4: $mejora \leftarrow$ falso;
5: **para todo** par $(i, j) : 1 \leq i \leq nK^* < j \leq n$ **hacer**
6: $K' \leftarrow$ `intercambiar`$(i, j, \pi, \pi^{-1}, w, K)$;
7: **si** $K' < K^*$ **entonces**
8: `actualizar-variables`$(i, j, \pi, \pi^{-1}, f, b, K, K^*, nK^*)$;
9: $mejora \leftarrow$ cierto;
10: **fin si**;
11: **fin para**;
12: **fin mientras**;
13: **devolver** (π, K^*);
14: **fin** Busqueda-local;

GRASP con *Path Relinking*

Los algoritmos tipo GRASP se basan en la construcción de distintas soluciones con un procedimiento constructivo voraz aleatorizado y en la mejora de dichas soluciones empleando

un procedimiento de búsqueda local. Esto permite explorar diferentes zonas del espacio de búsqueda. Además, GRASP ofrece una gran versatilidad, ya que es fácil de hibridar con otras metaheurísticas.

PR fue originalmente propuesto por Fred Glover [70, 73] en el contexto de *Tabu Search* como una estrategia de intensificación basada en la exploración de las trayectorias entre dos soluciones de alta calidad. Puede verse un pseudocódigo de PR en el Algoritmo 2.4. Inicialmente, el procedimiento recibe dos soluciones (x_s, x_t) y calcula su diferencia simétrica (paso 2). En cada iteración del algoritmo (pasos 5 a 13) se realiza un movimiento de modo que la solución de partida x_s se parezca más a la solución objetivo x_t. El movimiento escogido, de entre aquéllos posibles, será el que produzca una solución de mejor calidad (paso 6). La solución obtenida tras el movimiento (paso 8) será la nueva solución sobre la que seguir realizando movimientos. Cada solución generada es evaluada para determinar si es mejor que la mejor solución encontrada hasta el momento (pasos 9 a 12). El proceso finaliza cuando se alcanza la solución objetivo.

Algoritmo 2.4 Reencadenamiento de Trayectorias (extraído de [70]).

1: **procedimiento** Path-Relinking(x_s, x_t)
2: Calcular la diferencia simétrica $\Delta(x_s, x_t)$;
3: $f^* \leftarrow \min\{f(x_s), f(x_t)\}$;
4: $x^* \leftarrow \text{argmin}\{f(x_s), f(x_t)\}$;
5: $x \leftarrow x_s$;
6: **mientras** $\Delta(x_s, x_t) \neq 0$ **hacer**
7: $m^* \leftarrow \text{argmin}\{f(x \oplus m) : m \in \Delta(x_s, x_t)\}$;
8: $\Delta(x \oplus m^*, x_t) \leftarrow \Delta(x_s, x_t) \setminus \{m^*\}$;
9: $x \leftarrow x \oplus m^*$;
10: **si** $f(x) < f^*$ **entonces**
11: $f^* \leftarrow f(x)$;
12: $x^* \leftarrow x$;
13: **fin si**;
14: **fin mientras**;
15: **devolver** (x^*);
16: **fin** Path-Relinking;

Por otro lado, la idea de hibridar GRASP con PR se introdujo originalmente en [99] y consiste en mantener un conjunto de soluciones (denominado *elite set*) formado por soluciones de buena calidad creadas con GRASP. En esta versión de PR, cada iteración del algoritmo GRASP genera una nueva solución, que es empleada para aplicar la técnica de PR junto con una solución del *elite set* escogida aleatoriamente. Las nuevas soluciones generadas durante la etapa de PR serán evaluadas para entrar a formar parte del *elite set*.

EvoPR fue propuesto como una etapa posterior a GRASP con PR [157] en la cual el conjunto de soluciones que forman el *elite set* (población inicial P_0 formada por GRASP y PR) es evolucionado después de un cierto número de iteraciones, dando lugar a nuevas poblaciones

$(P_1, P_2, \ldots)$ de igual tamaño. Dicha evolución consiste en emplear la técnica de PR entre las soluciones del *elite set* y añadir nuevas soluciones a éste con los mismos criterios de entrada al *elite set* establecidos anteriormente. La evolución del *elite set* se detiene cuando no es posible añadir nuevas soluciones al mismo.

Se puede considerar PR como una extensión del mecanismo de combinación de soluciones de SS [73]. El método de combinación PR se basa en la generación de trayectorias entre soluciones, en el espacio de búsqueda, en lugar de llevar a cabo combinaciones lineales entre ellas.

Desde un punto de vista algorítmico, PR comparte con SS los cinco procedimientos generales expuestos en la Sección 1.4.2, de modo que PR y SS son métodos que operan sobre un conjunto de soluciones de referencia y difieren en la forma en la que el conjunto de referencia es construido, mantenido, actualizado y mejorado [78].

Algoritmo 2.5 GRASP con Reencadenamiento de Trayectorias Evolutivo [2].

procedimiento GRASP-EvoPR(i_{max}, j_{max})
$P \leftarrow \varnothing$;
$local \leftarrow 0$;
$global \leftarrow 0$;
mientras $global < i_{max}$ **hacer**
    mientras $local < j_{max}$ **hacer**
        $x \leftarrow$ `Construcción-Voraz-Aleatorizada`;
        $x \leftarrow$ `Búsqueda-Local`(x);
        Actualizar el *elite set* P con x;
        si $|P| > 0$ **entonces**
            Elegir aleatoriamente una solución $y \in P$;
            $x_p \leftarrow$ `Path`-Relinking (x, y);
            Actualizar el *elite set* P con x_p;
        fin si;
        $local \leftarrow local + 1$;
    fin mientras;
    $k \leftarrow 0$;
    repetir
        $k \leftarrow k + 1$;
        $P_k \leftarrow P$;
        $P_{k+1} \leftarrow \varnothing$;
        para algunos $\{x, y\} \in P_k \times P_k$ **hacer**
            $x_p \leftarrow$ `Path`-Relinking (x, y);
            Actualizar el *elite set* P_{k+1} con x_p;
        fin para algunos;
        $P \leftarrow P_{k+1}$;
    hasta que $\min\{f(x), x \in P_{k+1}\} \geq \min\{f(x), x \in P_k\}$;
    $local \leftarrow 0$;
    $global \leftarrow global + 1$;
fin mientras;
$x^* \leftarrow \text{argmin}\{f(x), x \in P)\}$;
devolver (x^*);
fin GRASP-EvoPR;

Los autores de este trabajo proponen construir soluciones y mejorarlas con los algoritmos constructivo y de búsqueda local propuestos más arriba e hibridar este esquema tipo GRASP con un algoritmo de Reencadenamiento de Trayectorias Evolutivo. En Algoritmo 2.5 se muestra el pseudocódigo de este algoritmo, que recibe como parámetro el número de iteraciones globales (i_{max}) y locales (j_{max}) que se ejecutará. Las iteraciones globales (pasos 5 a 30) determinan las veces que se repite el proceso de intensificación del *elite set*, mientras que las iteraciones locales (pasos 6 a 16) determinan cuántas soluciones se construirán, mejorarán y se les aplicará la técnica de *Path Relinking*, dentro de cada iteración global.

Capítulo 3

Algoritmos exactos para la resolución del CMP

La resolución exacta de un problema de optimización permite obtener soluciones óptimas al mismo. En este capítulo se repasan las diferentes técnicas exactas disponibles y se detallan los algoritmos propuestos, basados en la técnica de Ramificación y Acotación, para la resolución exacta del «problema de la minimización de la anchura de corte en ordenaciones lineales». Además, se exponen de manera detallada distintos recorridos posibles del árbol de exploración, así como el conjunto de cotas inferiores propuestas. Se realiza también un breve repaso de los métodos previos introducidos en el Capítulo 2 y, por último, se presenta un primer algoritmo heurístico, empleado para obtener una cota superior inicial al problema.

3.1. Introducción a los métodos de resolución exacta

Los problemas relacionados con la toma de decisiones han sido tradicionalmente abordados empleando distintos modelos clásicos de optimización. Un modelo de optimización es un modelo matemático (conjunto de expresiones que relacionan variables y valores) en el que se define una función a optimizar y cuyas variables están sujetas a un conjunto de restricciones. Estos modelos permiten representar problemas determinados y, mediante la resolución del modelo, obtener una solución al problema.

Los diferentes modelos de optimización, también conocidos como modelos de Programación Matemática, pueden ser clasificados atendiendo a diversos criterios. Uno de estos criterios hace referencia a la linealidad de las funciones que intervienen en el modelo, permitiendo así una clasificación en modelos lineales y modelos no lineales. La Programación Lineal (LP, del inglés *Linear Programming*) es una rama de la Programación Matemática dedicada a los modelos lineales. En LP, tanto la función objetivo como las restricciones son funciones lineales.

En referencia a cómo sean las variables de decisión del modelo LP éste puede ser a su vez categorizado como continuo, entero o mixto. Cuando las variables son continuas (toman valores

reales) se trata de un modelo continuo. Si las variables son discretas (toman valores enteros) se denomina entero (IP, del inglés *Integer Programming*). Un caso especial de IP es la formulación 0-1, en la que las variables sólo pueden tomar los valores cero o uno. Estas variables son conocidas como variables binarias. Por último, si las variables de decisión empleadas en el modelo pueden ser tanto continuas como discretas, se trata de un modelo mixto (MIP, del inglés *Mixed Integer Programming*).

Una solución a un modelo de programación lineal consiste en un conjunto de valores para las variables del mismo. Dichos valores deben ser consistentes con las ecuaciones o inecuaciones lineales que forman el modelo.

Para algunos de los modelos anteriormente descritos existen técnicas de resolución exacta eficientes, cuando el tamaño de la instancia es pequeño. Las principales técnicas exactas podrían categorizarse de la siguiente manera [172]:

- **Algoritmos enumerativos**: emplean árboles de búsqueda y están basados en la estrategia de divide y vencerás. El objetivo es crear particiones del espacio de búsqueda que permitan abordar problemas más pequeños contenidos en el problema original. Ejemplos de este tipo de técnicas son los algoritmos de Ramificación y Acotación, los Árboles A* y la Programación Dinámica.
- **Métodos de relajación y de descomposición**: las técnicas de relajación (como la relajación Lagrangiana) consisten en la eliminación de restricciones del modelo, permitiendo la obtención de cotas inferiores. Las técnicas de descomposición (como los métodos de descomposición de Bender) se basan en el establecimiento de valores para algunas de las variables del problema y en la resolución del modelo simplificado resultante.
- **Métodos basados en planos de corte**: están englobados dentro de la rama de la combinatoria poliédrica, y se basan en la poda del espacio de búsqueda. Su objetivo es tratar de encontrar relajaciones IP lo más ajustadas posibles.

Varios de los algoritmos anteriormente mencionados, podrían ser definidos como algoritmos de búsqueda basados en árboles de exploración.

A continuación, en las secciones 3.1.1, 3.1.2, 3.1.3 y 3.1.4 se repasan más detalladamente algunas de las técnicas anteriores.

3.1.1. Ramificación y Acotación

El algoritmo de Ramificación y Acotación, ya introducido en el Capítulo 1 está basado en la enumeración implícita de todas las soluciones posibles para el problema considerado. El espacio de soluciones se explora dinámicamente construyendo un árbol cuya raíz representa el espacio

de búsqueda completo, los nodos intermedios representan problemas más pequeños contenidos en el problema original y cada hoja es una potencial solución. El tamaño de los problemas en el árbol de exploración disminuye a medida que la exploración se acerca a las hojas.

La construcción del árbol y su exploración se realizan empleando dos operaciones básicas: ramificación y poda. El algoritmo se desarrolla en sucesivas iteraciones en las que la mejor solución encontrada se va actualizando progresivamente. Los nodos generados que aún no han sido ramificados se mantienen en una lista que inicialmente contendrá sólo la raíz. La estrategia de ramificación determina el orden en el que las distintas ramas son exploradas, mientras que la estrategia de poda, basada en las cotas al problema, elimina las soluciones parciales que no conducen a soluciones óptimas. El algoritmo termina cuando no quedan nodos pendientes de ser ramificados, bien porque ya han sido explorados, o bien porque han sido podados.

Los conceptos más importantes en el diseño eficiente de un algoritmo de Ramificación y Acotación, son las estrategias de ramificación y la calidad de las cotas empleadas. Tanto las cotas propuestas, como las estrategias de ramificación empleadas en esta Tesis Doctoral para abordar el CMP, son repasadas en la Sección 3.3.2 y en la Sección 3.3.3 respectivamente.

3.1.2. Programación Dinámica

La Programación Dinámica es una técnica consistente en la división recursiva de un problema en problemas más sencillos contenidos en el primero. Estos problemas, de menor tamaño, son resueltos de manera óptima y el coste de su solución almacenado para ser utilizado posteriormente en el cálculo del óptimo del problema inicial. El procedimiento consiste en tomar decisiones en etapas sucesivas, cada una de las cuales puede tener asociados uno o varios estados. Cuando se alcanza un estado determinado, éste contiene toda la información necesaria para tomar una decisión. Dicha decisión no depende de las decisiones que se hayan tomado en estados anteriores. La idea consiste en que el algoritmo parta de las últimas decisiones, es decir, del estado en el que se resuelve el problema más sencillo, con el objetivo final de optimizar el coste total de las distintas decisiones necesarias para resolver el problema inicial [142].

La Programación Dinámica se basa en el principio de Bellman [11] que enuncia que «*cuando se dispone de una secuencia óptima de decisiones, toda secuencia menor contenida en la primera es a su vez óptima*». Este principio no es siempre aplicable, por lo que es necesario verificar que se cumple para cada problema determinado. Para ello, debe ser posible alcanzar la solución al problema a través de una secuencia de decisiones.

Para diseñar un procedimiento de Programación Dinámica es necesario definir los siguientes componentes [14]:

- División del problema en etapas y estados. Un problema puede ser dividido en un número de etapas k y, a su vez, cada etapa tener asociada un número determinado de estados u_k.

- Definir el coste de la etapa inicial.
- Definir la relación recursiva para un estado en la etapa k de acuerdo con los estados de etapas anteriores.

Empleando Programación Dinámica se evita enumerar el espacio de búsqueda entero, podando secuencias de decisiones parciales que no conducen a una solución óptima. Esta técnica presenta similitudes con la técnica de Ramificación y Acotación descrita en la Sección 3.1.1 en el sentido de que realiza una enumeración inteligente de las posibles soluciones al problema.

3.1.3. Ramificación y Corte

La técnica de Ramificación y Corte, más conocida por su nombre en inglés como «*Branch and Cut*», es una técnica exacta aplicada a problemas de optimización formulados en IP. Está basada en la combinación de la técnica de Ramificación y Acotación (descrita en la Sección 3.1.1) con los Planos de Corte. Para ello, se aplican Planos de Corte en cada uno de los nodos del árbol de exploración generados por el algoritmo de Ramificación y Acotación [91].

Los Planos de Corte fueron propuestos por Gomory en 1958 [76] y consisten en añadir restricciones, de manera iterativa, a la formulación IP de un problema. La idea es partir de un conjunto pequeño de restricciones y calcular el óptimo para el problema empleando esas restricciones. A continuación, se revisa si alguna de las restricciones que no se encuentran en el nuevo modelo, pero que estaban en el modelo original, no han sido satisfechas. En tal caso, se procede a añadir una nueva restricción para que ésta se cumpla y se resuelve el modelo nuevamente. El proceso se repite hasta que todas las restricciones sean satisfechas.

3.1.4. Métodos de relajación

Además de los métodos mencionados anteriormente, existen otras técnicas capaces de relajar algún requisito del problema. En general, estas relajaciones se llevan a cabo eliminando una restricción del problema o sustituyéndola por otra más fácil de satisfacer [172]. Los dos mecanismos de relajación más utilizados son la Relajación LP y la Relajación Lagrangiana.

Relajación LP

Consiste en ignorar directamente alguna de las restricciones de integridad de un problema. Una vez eliminada la restricción, se resuelve el modelo resultante, por ejemplo, con un *solver*. El resultado obtenido es una cota inferior al problema. En el caso de que dicha solución satisfaga todas las restricciones originales del problema, se trataría de una solución óptima. Esta técnica es comúnmente utilizada junto con algoritmos de Ramificación y Acotación [172].

Relajación Lagrangiana

Este tipo de relajación se emplea para generar cotas inferiores ajustadas a problemas de optimización. La idea principal consiste en eliminar algunas restricciones del problema y añadirlas a la función objetivo. Para cada restricción eliminada, se añade una función de penalización. En el diseño de este tipo de relajación es esencial la selección de la restricción a relajar [172].

3.1.5. *Solver*

Los *solver* son librerías o aplicaciones capaces de resolver problemas de optimización expresados mediante modelos matemáticos. Para ello hacen uso de diferentes técnicas en función del tipo de modelo que traten de resolver. Cuando se trata de problemas lineales existen algoritmos muy eficientes, tales como el método «Simplex» [43] o el método del «Punto Interior» [94]. Por otro lado, para modelos IP o MIP, algunas de las técnicas más utilizadas son los Planos de Corte y la técnica de Ramificación y Acotación. Algunos *solver* disponen también de optimizadores locales para modelos no lineales.

Actualmente hay un gran número de *solver* disponibles, tanto privativos, como libres. En el Apéndice B se repasan algunos de los *solver* más relevantes, entre los que se encuentran los descritos a continuación:

- **CPLEX**[1]: es quizá el *solver* privativo más conocido en la actualidad. Se trata de un *solver* de alto rendimiento capaz de abordar problemas con gran cantidad de variables. Soporta diferentes tipos de modelos de Programación Matemática: LP, MIP y también modelos de Programación Cuadrática (QP, del inglés *Quadratic Programming*). Además, posee distintas interfaces para facilitar la implementación de los modelos en C++, C#, Java y Python. Las últimas versiones incorporan soporte para aprovechar las ventajas de los procesadores *multicore*.

- **Gurobi**[2]: se trata de un *solver* privativo que soporta modelos LP, MIP y QP. Incluye implementaciones de alto rendimiento de los métodos: «Simplex» [43], «Dual Simplex» [104] y «Parallel Barrier». Para los modelos mixtos, tanto de LP como de QP, incorpora métodos basados en Planos de Corte y Heurísticas. Todos los métodos anteriores pueden hacer uso de la técnica de *presolving*, capaz de simplificar el modelo de entrada. Entre sus principales características cabe destacar también, que está pensado desde sus orígenes para explotar las ventajas de los procesadores *multicore*. El procesamiento paralelo se comporta de manera determinista, es decir, dos ejecuciones independientes del mismo modelo sobre

[1] `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`
[2] `http://www.gurobi.com/`

una misma instancia de entrada, obtendrán el mismo resultado. Proporciona interfaces para Python, C, C++, Java y lenguajes .NET.

- **Xpress-MP**[3]: es un *solver* privativo de alto rendimiento para LP, MIP y QP mixta (MIQP). Al igual que CPLEX y Gurobi, Xpress-MP utiliza las técnicas de «Simplex», Ramificación y Acotación, Planos de Corte, así como distintas heurísticas para la resolución de los modelos.

- **GLPK**[4]: *solver* libre, cuyo acrónimo procede del inglés *GNU Linear Programming Kit*. Está diseñado para LP y MIP. Emplea los métodos del «Simplex», «Punto Interior» y Ramificación y Corte, entre otros. Se distribuye como un conjunto de rutinas escritas en ANSI C y organizadas en forma de librería.

- **LP_Solve**[5]: *solver* libre distribuido bajo la licencia LGPL[6] (del inglés, *Lesser General Public License*) pensado para la resolución de modelos LP y MIP. Está basado en el método «Simplex» y en la técnica de Ramificación y Acotación.

Disponer de una formulación IP para el CMP permite medir la calidad de los resultados obtenidos por los algoritmos de Ramificación y Acotación propuestos en esta Tesis Doctoral, al compararlos con los resultados obtenidos por un *solver* empleando dicha formulación. En concreto, se ha empleado la formulación IP propuesta en [114] para el CMP, generando un modelo para el *solver* CPLEX. En el Capítulo 5 se muestra la comparación entre los resultados obtenidos por los algoritmos de Ramificación y Acotación propuestos y la formulación IP del problema resuelta mediante el *solver* CPLEX.

3.2. Métodos previos

En el Capítulo 2 de esta Tesis Doctoral se lleva a cabo un repaso de los trabajos previos relacionados con el CMP. En cuanto a la resolución exacta del problema se refiere, es importante destacar que, pese al conocido interés práctico del mismo [1, 25, 89, 93, 135, 153] no se han encontrado algoritmos exactos (en concreto basados en la técnica de Ramificación y Acotación) que aborden su resolución para grafos genéricos. No obstante, cabe mencionar un trabajo en el que se propone una posible formulación IP para el CMP [114], que se resume en la Sección 3.2.1.

3.2.1. Representación en Programación Entera del CMP

Como se ha indicado anteriormente, el CMP puede ser expresado como un problema de Programación Entera. Una explicación detallada de la formulación IP propuesta en [114] puede

[3] http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx
[4] http://www.gnu.org/software/glpk/
[5] http://lpsolve.sourceforge.net/
[6] http://www.gnu.org/licenses/lgpl.html

encontrarse en la Sección 2.3. A continuación, se resume dicha formulación:

Minimizar S,

sujeto a:

$$X_i^k \in \{0,1\},\ \forall\, i,k \in \{1,\ldots,n\} \tag{3.1}$$

$$\textstyle\sum_{k\in\{1,\ldots,n\}} X_i^k = 1,\ \forall i \in \{1,\ldots,n\} \tag{3.2}$$

$$\textstyle\sum_{i\in\{1,\ldots,n\}} X_i^k = 1,\ \forall k \in \{1,\ldots,n\} \tag{3.3}$$

$$Y_{i,l}^{k,j} \leq X_i^k \tag{3.4}$$

$$Y_{i,l}^{k,j} \leq X_l^j \tag{3.5}$$

$$X_i^k + X_l^j \leq Y_{i,l}^{k,j} + 1 \tag{3.6}$$

$$\textstyle\sum_{(k\leq c<l)\vee(l\leq c<k)} Y_{i,l}^{k,j} \leq S,\ \forall c \in \{1,\ldots,n-1\} \tag{3.7}$$

La formulación anterior fue probada por los autores del trabajo, empleando el *solver* MINLP[7] (*solver* para modelos MIP que implementa un algoritmo basado en la técnica de Ramificación y Acotación). Utilizando dicho *solver* fueron capaces de resolver de manera óptima el CMP para grafos tipo malla[8] de hasta 6 vértices.

En esta Tesis Doctoral se ha empleado esta formulación para generar un modelo para el *solver* CPLEX capaz de resolver instancias de tamaño mucho mayor al mencionado por los autores de la misma. La adaptación de la formulación propuesta al modelo de CPLEX se ha llevado a cabo empleando la API para Java del propio *solver*.

3.3. Algoritmos de Ramificación y Acotación para la resolución del CMP

En esta Tesis Doctoral se proponen varios algoritmos basados en la técnica de Ramificación y Acotación (descrita en la Sección 3.1.1) para la resolución exacta del CMP. Como se indicó anteriormente, la eficiencia de un algoritmo de Ramificación y Acotación depende en gran medida de la calidad de las cotas propuestas, así como de las estrategias de ramificación seguidas.

A continuación, en la Sección 3.3.1 se detallan aspectos relevantes del árbol de exploración para el CMP. En la Sección 3.3.2 se presentan las distintas cotas inferiores propuestas para el algoritmo de Ramificación y Acotación. En la Sección 3.3.3 se muestran diferentes recorridos del árbol de exploración, destacando las ventajas e inconvenientes de cada uno de ellos, y detallando los recorridos empleados en esta Tesis Doctoral para abordar el CMP. Por último, en la Sección 3.3.4 se estudia cómo almacenar el árbol de exploración en memoria.

[7] `http://www.neos-server.org/`

[8] Puede encontrarse una definición de este tipo de grafos en la sección 2.1.7

3.3.1. El árbol de exploración

En optimización, se denomina árbol de exploración (o árbol de búsqueda) a la representación del espacio de soluciones de un problema determinado, mediante una estructura de datos tipo árbol. La raíz del árbol representa el espacio de búsqueda completo y, cada nodo hoja, una solución concreta. El árbol tiene tantas hojas como soluciones posibles, por lo tanto, el conjunto de todas las hojas compone el espacio de soluciones. Los nodos intermedios representan soluciones parciales o incompletas, cuyos hijos serán el subconjunto de soluciones (completas o incompletas) que son alcanzables desde ese nodo.

El árbol de exploración se construye a partir de la raíz, empleando ramificaciones. Una ramificación consiste en una división del espacio de soluciones S en subconjuntos menores ($S' \subset S$). El proceso de ramificación se aplica en cada nuevo subconjunto generado, hasta crear una estructura de árbol de manera que represente todo el espacio de soluciones.

En la Figura 3.1 se puede ver un ejemplo de árbol de exploración para un problema de ordenaciones lineales, como es el CMP. Este árbol representa el espacio de soluciones de un grafo $G(V, E)$ con tres vértices, $V = \{A, B, C\}$. En el nivel 0 del árbol se encuentra la raíz del mismo, donde ninguna posición de la ordenación lineal (f) tiene un vértice asignado. Por el contrario, en el nivel 3, todas las posiciones de f tienen asignado un vértice de G. En este nivel se encuentran las hojas del árbol y, por lo tanto, todas las posibles soluciones al problema, para un grafo de 3 vértices. El resto de nodos del árbol son los nodos intermedios. Estos nodos contienen una solución incompleta, en la que algunas posiciones de f tienen un vértice asignado y otras posiciones no.

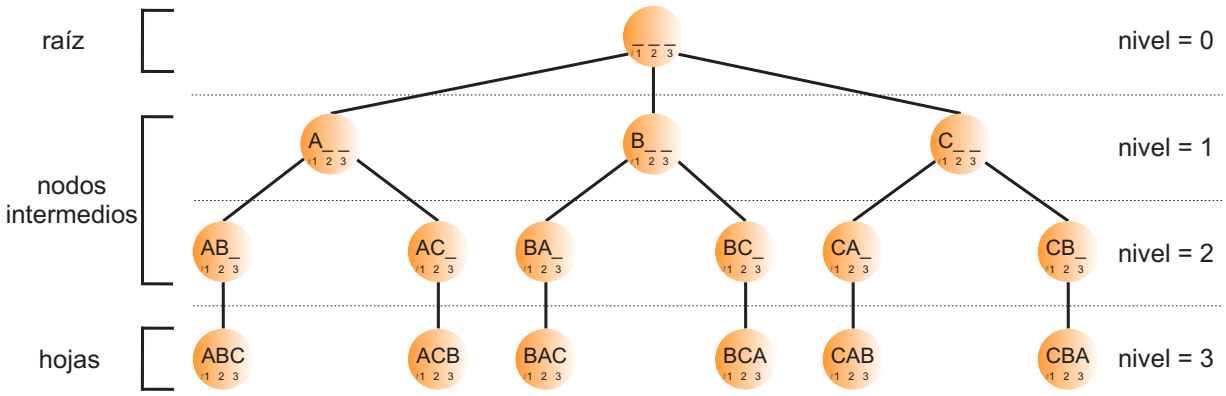


Figura 3.1: Ejemplo de árbol de exploración para el CMP para un grafo de 3 vértices.

Número de nodos del árbol de exploración para el CMP

El objetivo del CMP consiste en encontrar una ordenación de los vértices de un grafo $G(V, E)$ con $|V| = n$. Dado G, el número de ordenaciones diferentes de sus vértices asciende a $n!$

ordenaciones. Este número es, a su vez, el número de soluciones al problema, lo que traducido a un árbol de exploración, implica que el árbol debe tener $n!$ hojas para un grafo con n vértices. Pese a que el número de soluciones completas (representadas por las hojas del árbol) es ya de por sí muy elevado, el tamaño del árbol de exploración es aún mayor, ya que hay que considerar, además de los nodos hojas, los nodos intermedios y la raíz, que contienen soluciones incompletas. El número de nodos (N) de un árbol de exploración completo para el CMP es:

$$N = \sum_{i=0}^{n} \frac{n!}{(n-i)!}$$

donde i es el nivel del árbol, luego $i = 0$ representa a la raíz, mientras que $i = n$ representa a los nodos hoja.

El motivo por el que problemas como el CMP se consideran intratables desde el punto de vista computacional se puede intuir al observar la Tabla 3.1. En dicha tabla se muestra el número de soluciones posibles para distintos tamaños del grafo que representa al problema de entrada. Asimismo, se muestra el número de nodos que tendría el árbol de exploración completo para el mismo tamaño. Pese a que podría considerarse que los grafos representados en la Tabla 3.1 son de tamaño pequeño, para alguno de ellos sería prácticamente imposible enumerar todas sus soluciones en un tiempo razonable. Por ello, es sencillo imaginar la dificultad que entrañaría resolver problemas de tamaño aún mayor.

n	**Número de soluciones**	**Nodos del árbol**
5	$1{,}20000 \times 10^{2}$	$3{,}26000 \times 10^{2}$
10	$3{,}62880 \times 10^{6}$	$9{,}86410 \times 10^{6}$
20	$2{,}43290 \times 10^{8}$	$6{,}61331 \times 10^{18}$
40	$8{,}15915 \times 10^{47}$	$2{,}21789 \times 10^{48}$
80	$7{,}15695 \times 10^{118}$	$1{,}94550 \times 10^{119}$
160	$4{,}71472 \times 10^{284}$	$1{,}28160 \times 10^{285}$

Tabla 3.1: Relación entre el número de vértices (n) del grafo de entrada, el número de soluciones al CMP y el número de los nodos del árbol de exploración.

A priori, encontrar una solución óptima para el CMP implicaría examinar, de manera exhaustiva, todas las soluciones del árbol de exploración. No obstante, la técnica de Ramificación y Acotación permite evitar la exploración de ciertas zonas del árbol que se consideran poco prometedoras, es decir, de las cuales se tiene la certeza de que no alcanzarán una solución óptima. Para ello, hace uso de las denominadas funciones de cota, que son estudiadas en la Sección 3.3.2. El resultado de evitar la exploración de ciertas zonas del espacio de soluciones, es un algoritmo más eficiente que permite abordar problemas de mayor tamaño en el mismo tiempo.

3.3.2. Cotas inferiores para soluciones parciales

En esta Tesis Doctoral se proponen un conjunto de funciones de cota, capaces de calcular cotas inferiores para el CMP. Éstas, serán empleadas junto con los recorridos del árbol de exploración descritos en la Sección 3.3.3, para conformar diferentes algoritmos de Ramificación y Acotación.

Según el Diccionario de la Lengua de la Real Academia Española, en matemáticas, una cota se define como: «*elemento de un conjunto que limita, inferior o superiormente, los elementos de la sucesión de un subconjunto*» [152]. En optimización, el término cota hace referencia a un valor que garantiza que la solución óptima a un problema de optimización está por encima (si la cota es inferior) o por debajo (si la cota es superior) de dicho valor.

Dado un nodo de un árbol de exploración, una función de cota permite realizar una estimación de lo buena que puede llegar a ser cualquier solución descendiente de ese nodo, sin necesidad de expandir la rama correspondiente hasta alcanzarla. Es decir, dada una solución parcial, una función de cota da como resultado una cota, que certifica que el valor de la función objetivo de cada una de las soluciones alcanzables desde ese nodo es, como mucho, tan bueno como ese valor.

Cuando se está empleando un algoritmo de Ramificación y Acotación para buscar soluciones óptimas a un problema de optimización, los valores de cota ayudan a evitar la expansión de zonas del árbol de exploración poco prometedoras, mediante el proceso conocido como poda. La poda de una rama del árbol de exploración consiste en descartar todas las soluciones que se encuentran por debajo de un nodo determinado en el árbol.

Para llevar a cabo una poda en un árbol de exploración, que representa a un problema de minimización, es necesario disponer de una cota superior (UB, del inglés *Upper Bound*) que se corresponde con la mejor solución encontrada al problema hasta ese momento. En cada nodo del árbol que esté siendo explorado, se calcula una cota inferior (LB, del inglés *Lower Bound*) que estime lo buenas que pueden llegar a ser las soluciones alcanzables desde ese nodo. Si la cota inferior de un nodo es mayor o igual que la cota superior, no será necesario expandir ese nodo, ya que se tiene la certeza de que la mejor solución alcanzable desde él (cota inferior) no será mejor que la mejor solución encontrada previamente (cota superior).

Evaluación de una solución parcial

Como se ha indicado anteriormente, las funciones de cota se emplean para realizar estimaciones sobre soluciones parciales del árbol de exploración. Además de las estimaciones, también es posible evaluar la parte de la solución que se tiene ya construida. Este valor, también es una cota inferior en el caso concreto del CMP, en el nodo que contiene dicha solución parcial, ya que todos los nodos descendientes del nodo evaluado tendrán una solución, como mucho, tan

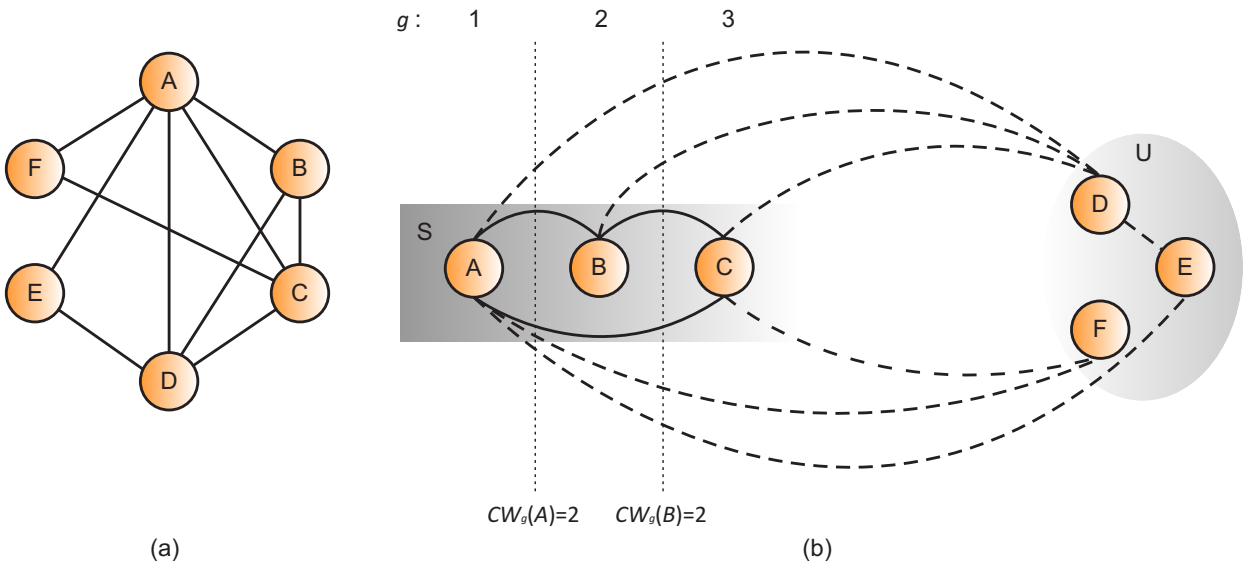


Figura 3.2: (a) Representación de un grafo $G(V, E)$. (b) Solución parcial o incompleta de los vértices de G.

buena como el valor de la solución parcial de ese nodo.

Dado un grafo $G(V, E)$, una solución parcial se define como el par (S, g) siendo S un subconjunto de los vértices de V ($S \subset V$) con k vértices ($k < n$) y g una ordenación tal que $g \in \Pi_k$ que asigna los enteros $\{1, 2, \ldots, k\}$ a los vértices de S. A partir de la solución parcial anterior, una solución completa para el CMP se podría obtener añadiendo $n - k$ elementos de $V \setminus S$ a S, y asignándoles los enteros $\{k + 1, k + 2, \ldots, n\}$. Por lo tanto, los elementos en S ordenados de acuerdo a g pueden ser vistos como una solución parcial o incompleta sobre G para el CMP.

En la Figura 3.2.b se muestra una solución parcial (S, g) del grafo de la Figura 3.2.a, donde los vértices de $S = \{A, B, C\}$ han sido etiquetados en g ($g(A) = 1$, $g(B) = 2$, $g(C) = 3$) y los vértices D, E y F permanecen sin etiquetar en el conjunto U. Tomando como base esta solución parcial, podrían construirse las siguientes soluciones completas: $S_g = \{(A, B, C, D, E, F), (A, B, C, D, F, E), (A, B, C, F, D, E), (A, B, C, F, E, D), (A, B, C, E, D, F), (A, B, C, E, F, D)\}$.

Se podría decir que la solución parcial (S, g) representada en la Figura 3.2.b, con $S \subset V$ y $g \in \Pi_k$, es una solución completa respecto del grafo $G_S = (S, E_S)$ que está formado por el conjunto de vértices etiquetados S y el conjunto de aristas que unen dichos vértices $E_S \subset E$. En concreto, el grafo de la Figura 3.2.b, $G_S = (S, E_S)$, se corresponde con el conjunto de vértices $S = \{A, B, C\}$ y el conjunto de aristas $E_S = \{(A, B), (B, C), (A, C)\}$.

Una vez definida una solución parcial, se puede particularizar la expresión genérica (mostrada en el Capítulo 1) que se utiliza para calcular el *cutwidth* de una ordenación f de manera que pueda ser aplicada a una solución parcial (S, g). Para ello, es necesario definir el *cutwidth* de

cada vértice en G_S con respecto a la ordenación g, considerando únicamente las aristas en E_S:

$$CW_g(v) =\mid \{(u,v) \in E_S : g(u) \leq g(v) < g(w)\} \mid$$

En el ejemplo de la Figura 3.2.b el valor del *cutwidth* para cada vértice de S, según la expresión anterior, sería: $CW_g(A) = 2$, $CW_g(B) = 2$ y $CW_g(C) = 0$. Es evidente que los valores del *cutwidth* en la solución parcial proporcionan una cota inferior de los valores del *cutwidth* en cualquier solución completa $f \in S_g$. Por ejemplo, si se asignan las etiquetas 4, 5 y 6 a los vértices D, E y F respectivamente, se tendría que $CW_f(A) \geq CW_g(A) = 2$, $CW_f(B) \geq CW_g(B) = 2$ y $CW_f(C) \geq CW_g(C) = 0$. Por lo tanto, se puede concluir que el *cutwidth* de una ordenación completa f de los vértices de G, denotado por $CW_f(G)$ es mayor que el máx$\{CW_g(A), CW_g(B), CW_g(C)\} = 2$ y, a su vez, que este máximo es una cota inferior para el CMP. En términos matemáticos, para cualquier $f \in S_g$, el valor de la cota inferior basada en el *cutwidth* de la solución parcial y denotada por $LB(S, g)$, cumple:

$$CW_f(G) \geq LB(S,g) = \max_{v \in S} CW_g(v)$$

Además de la función de cota trivial previamente descrita para el CMP, se proponen otras cinco funciones de cota para el cálculo de cotas inferiores sobre soluciones parciales, que son detalladas a continuación.

Cota inferior 1 (LB_1)

La función de cota LB_1 proporciona una cota inferior para el CMP, basándose en el grado del vértice de mayor adyacencia de un grafo $G(V, E)$.

Sea $N(v)$ el conjunto de vértices adyacentes al vértice v, y $E(v)$ las aristas con un extremo en v. Considérese u como el vértice que precede a v en la ordenación f, es decir, aquél que está situado en la posición $f(v) - 1$. Si una arista en $E(v)$ es adyacente a un vértice w tal que $f(w) < f(v)$, entonces dicha arista contribuye en el *cutwidth* del vértice u, $CW_f(u)$. Si por el contrario, $f(w) > f(v)$, ésta sería computada en $CW_f(v)$. Entonces se satisface que $CW_f(u) + CW_f(v) \geq \mid N(v) \mid$, por lo tanto:

$$\text{máx}\{CW_f(u), CW_f(v)\} \geq \mid N(v) \mid /2$$

Considerando que el *cutwidth* de un grafo $CW_f(G)$ es el máximo de los *cutwidth* de todos sus vértices, se puede concluir que $\mid N(v) \mid /2$ es una cota inferior de $CW_f(G)$.

$$CW_f(G) \geq LB_1 = \max_{v \in V} \left\lceil \frac{\mid N(v) \mid}{2} \right\rceil$$

Dado que la estructura de la solución para el CMP es una ordenación, para todo vértice v situado en una ordenación f, sus adyacentes podrán estar colocados a derecha e izquierda del mismo. De manera intuitiva, se puede deducir que si sólo se considerara un vértice y sus adyacentes, la posición ideal para que los adyacentes de v generen un menor *cutwidth*, es que se sitúen la mitad a la derecha de la posición $f(v)$ y la mitad a la izquierda. Considerando uno a uno todos los vértices del grafo, el vértice de mayor grado sería el que generara un *cutwidth* mayor que el resto, colocando sus adyacentes en la mejor disposición posible. Pese a que al *cutwidth* de cada vértice en la ordenación se verá afectado por el resto de aristas del grafo, el *cutwidth* de la ordenación nunca será menor que $\left\lceil \frac{|N(v)|}{2} \right\rceil$ siendo v el vértice de mayor grado.

Por ejemplo, en el grafo de la Figura 3.2.a el vértice de mayor grado es el vértice A, que tiene un total de 5 adyacentes. En la Figura 3.3 se han representado las diferentes posiciones relativas del vértice A respecto a sus adyacentes. Como se puede observar, las ordenaciones representadas en las figura 3.3.c y 3.3.d son las que tienen un menor valor del *cutwidth* (considerando sólo los vértices del grafo adyacentes a A y las aristas con un extremo en A). En dichas ordenaciones, los adyacentes del vértice A están repartidos a la derecha e izquierda del mismo de manera equitativa (nótese que al tener un número impar de vértices adyacentes, uno de los lados tiene un adyacente más). Por el contrario, la Figura 3.3.a y la Figura 3.3.f representan las ordenaciones con un mayor valor del *cutwidth*, debido a que todos los adyacentes de A se encuentran a la derecha de A (Figura 3.3.2.a) o a su izquierda (Figura 3.3.2.f). Por lo tanto, en el mejor de los casos, el $CW_f(G)$ en dicho ejemplo, será igual a 3, ya que el vértice de mayor adyacencia es el vértice A y $\left\lceil \frac{|N(A)|}{2} \right\rceil = 3$.

Cota inferior 2 (LB_2)

La función de cota LB_2 determina el valor del *cutwidth* que tendrán los vértices etiquetados en una solución parcial cuando se complete dicha solución, sea cual sea el orden en el que se etiqueten los vértices que aún quedan por etiquetar. Está basada en las aristas que tiene cada vértice etiquetado, tomando en consideración tanto las aristas que conducen a otro vértice etiquetado (como se hacía en la evaluación de la solución parcial) como aquéllas que conducen a uno no etiquetado.

Dada una solución parcial (S, g) y una solución completa f en S_g, el *cutwidth* de un vértice $v \in S$ con respecto a f, denotado por $CW_f(v)$, puede ser calculado como:

$$CW_f(v) = CW_g(v) + \sum_{\substack{u \in S \\ 1 \leq g(u) \leq g(v)}} \mid N_U(u) \mid \tag{3.8}$$

donde $N_U(u)$ es el conjunto de vértices adyacentes a u que todavía no han sido etiquetados.

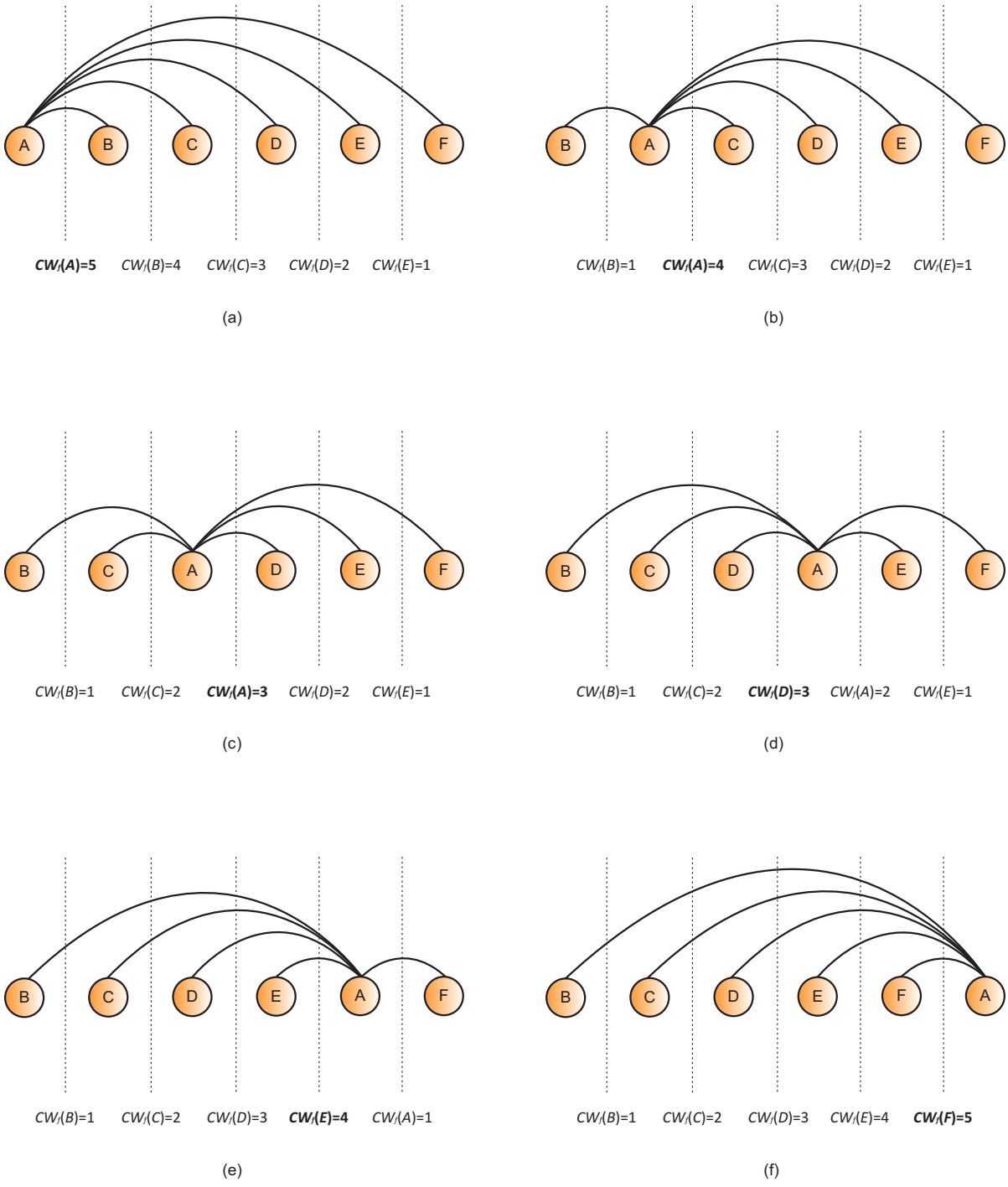


Figura 3.3: Posibles posiciones relativas del vértice A del grafo de la Figura 3.2, respecto a sus adyacentes.

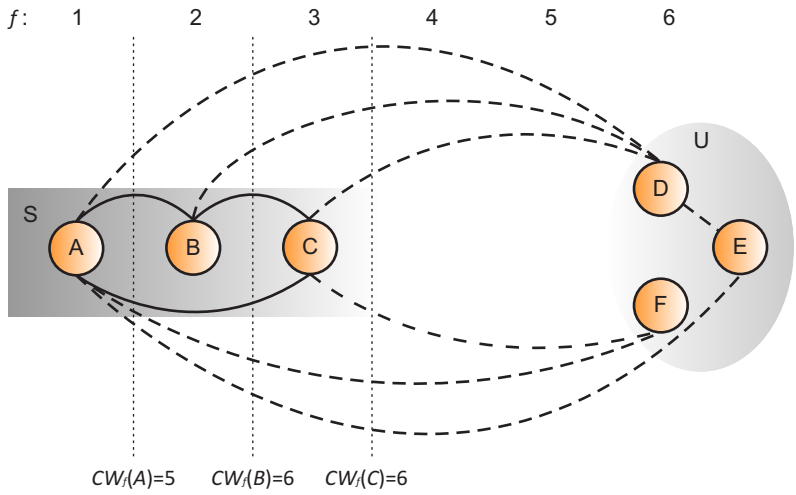


Figura 3.4: Valor del *cutwidth* que tendrá cada vértice etiquetado de una solución parcial, cuando ésta se complete.

El primer término de la expresión, $CW_g(v)$ corresponde al *cutwidth* de v en $G_S = (S, E_S)$. El segundo término computa el número de aristas con un extremo en un vértice etiquetado u, con $g(u) \leq g(v)$, es decir, previo a v en la ordenación g, y el otro extremo en un vértice no etiquetado w.

Es importante destacar que $f(w) > g(v)$ para cualquier $w \in U$ y cualquier solución f en S_g, ya que por este motivo se pueden computar todas las aristas con un extremo en un vértice no etiquetado, w, en el cálculo de $CW_f(v)$.

Dado que (1.9) proporciona una expresión para calcular $CW_f(v)$ para todo v en $S \subseteq V$ y $CW_f(G)$ es el máximo $CW_f(v)$ para todo v en V, se podría concluir que:

$$CW_f(G) \geq LB_2 = \max_{v \in S} \left\{ CW_g(v) + \sum_{\substack{u \in S \\ 1 \leq g(u) \leq g(v)}} \mid N_U(u) \mid \right\}$$

En la Figura 3.4 se muestra, para cada uno de los vértices etiquetados en una solución parcial, el valor del *cutwidth* que tendrán en cualquier solución completa f en S_g, creada a partir de la primera. Dicha solución completa satisfará:

$$CW_f(G) \geq \max\{CW_f(A), CW_f(B), CW_f(C)\} = \max\{5, 6, 6\} = 6$$

donde:

$$CW_f(A) = CW_g(A) + \mid N_U(A) \mid = 2 + 3 = 5$$
$$CW_f(B) = CW_g(B) + \mid N_U(A) \mid + \mid N_U(B) \mid = 2 + 3 + 1 = 6$$
$$CW_f(C) = CW_g(C) + \mid N_U(A) \mid + \mid N_U(B) \mid + \mid N_U(C) \mid = 0 + 3 + 1 + 2 = 6$$

Cota inferior 3 (LB_3)

La función de cota LB_3 está centrada en los vértices no etiquetados de una solución parcial, empleando algunas de las ideas descritas para LB_1 y LB_2. Dada una solución parcial (S, g) y un vértice no etiquetado $u \in U$, se denota por $N_S(u)$ al conjunto de vértices adyacentes a u que están etiquetados. Sea v_k el vértice en S con la mayor etiqueta, tal que $g(v_k) = k = \mid S \mid$. Para cualquier f en S_g y cualquier v en S, se verifica que $f(v) \leq f(v_k) < f(u)$. Entonces, se puede afirmar que $CW_f(v_k) \geq \mid N_S(u) \mid$. Además, junto con esta idea, también se puede aplicar a cada vértice no etiquetado ($u \in U$) el argumento descrito para LB_1, obteniendo así una cota mejorada, denominada LB_3:

$$CW_f(G) \geq LB_3 = \max_{u \in U} \left\{ \left\lceil \frac{\mid N(u) \mid}{2} \right\rceil, \; N_S(u) \right\}$$

En el ejemplo de la Figura 3.4, el valor de LB_3 para los vértices D, E y F sería:

$$LB_3(D) = \text{máx} \left\{ \left\lceil \frac{\mid N(D) \mid}{2} \right\rceil, \; N_S(D) \right\} = \text{máx}\{2, 3\} = 3$$

$$LB_3(E) = \text{máx} \left\{ \left\lceil \frac{\mid N(E) \mid}{2} \right\rceil, \; N_S(E) \right\} = \text{máx}\{1, 1\} = 1$$

$$LB_3(F) = \text{máx} \left\{ \left\lceil \frac{\mid N(F) \mid}{2} \right\rceil, \; N_S(F) \right\} = \text{máx}\{1, 2\} = 2$$

Cota inferior 4 (LB_4)

Como en el caso anterior, sea (S, g) una solución parcial, $u \in U$ un vértice no etiquetado, v_k el vértice en S con la mayor etiqueta, tal que $g(v_k) = k = \mid S \mid$ y f en S_g una solución completa. Si el vértice u es etiquetado en f con la etiqueta $k + 1$, es decir, u sigue al vértice v_k en la ordenación f, su *cutwidth* puede ser calculado como:

$$CW_f(u) = CW_g(v_k) - (\mid N_S(u) \mid - \mid N_U(u) \mid)$$

donde $N_S(u)$ es el conjunto de vértices adyacentes a u previamente etiquetados y $N_U(u)$ es el conjunto de vértices adyacentes a u que todavía no han sido etiquetados. Se puede calcular un

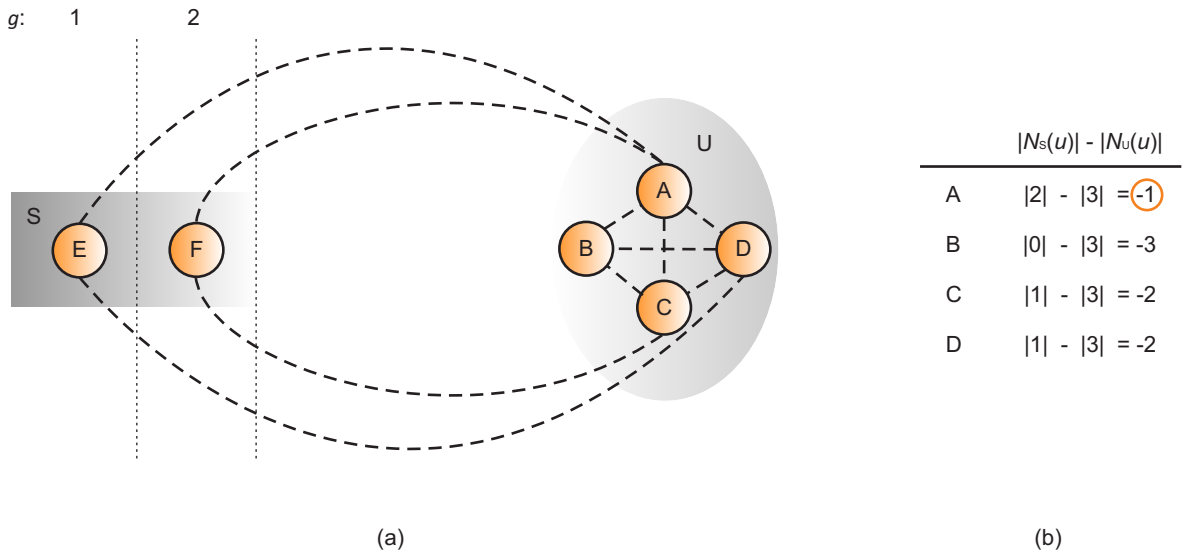


Figura 3.5: (a) Solución parcial del grafo de la Figura 3.2.a. (b) Balance entre adyacentes etiquetados y no etiquetados de cada vértice no etiquetado.

cota inferior del valor del *cutwidth* para un vértice que ocupe la posición $k+1$, calculando el máximo del término $\mid N_S(u) \mid - \mid N_U(u) \mid$ para todo $u \in U$. De manera resumida:

$$CW_f(G) \geq LB_4 = CW_g(v_k) - \max_{u \in U}(\mid N_S(u) \mid - \mid N_U(u) \mid)$$

En la Figura 3.5.a se muestra una solución parcialmente construida donde $S = \{E, F\}$, $g(E) = 1$, $g(F) = 2$ y $U = \{A, B, C, D\}$ con $CW_g(F) = 4$. En la Figura 3.5.b, se muestra la evaluación de cada posible candidato a ocupar la posición $k+1 = 3$. Para cada vértice no etiquetado, se calcula el balance entre número de adyacentes etiquetados y número de adyacentes sin etiquetar ($\mid N_S(u) \mid - \mid N_U(u) \mid$). De acuerdo con la definición dada previamente, se selecciona el vértice que tenga un mayor balance entre etiquetados y no etiquetados como el vértice más prometedor, de todos los que pueden ser colocados en la posición $k+1$. En este caso se selecciona el vértice A, dando un valor de $LB_4 = 4 - (-1) = 5$. Esto significa que, con independencia de qué vértice de U se etiquete en la siguiente iteración, el valor de la solución completa será mayor o igual a 5.

Cota inferior 5 (LB_5)

Dado un grafo $G(V, E)$ con $\mid V \mid = n$ y $\mid E \mid = m$, la función de cota LB_5 está basada en el cálculo del *cutwidth* de un grafo auxiliar G' con el mismo número de vértices y aristas que G, pero con una estructura que permite obtener el *cutwidth* mínimo para ese número de vértices y aristas. De este modo, el *cutwidth* de G' es una cota inferior del *cutwidth* de G, para cualquier etiquetado de los vértices de G. De hecho, se puede afirmar que el *cutwidth* de G', es una cota inferior para cualquier grafo con n vértices y m aristas.

En el caso de que $m < n$, se podría construir el grafo auxiliar G' a modo de *path* (ver Figura 3.6) en el que serían necesarias $m = n - 1$ aristas, para que se tratase de un *path* conectado. Para este ejemplo, el *cutwidth* de G' sería igual a 1. Es evidente, que si se pudiese generar este grafo G' con los vértices y aristas de G, cualquier ordenación f de los vértices de G tendría un *cutwidth* igual o mayor que $CW(G') = 1$.

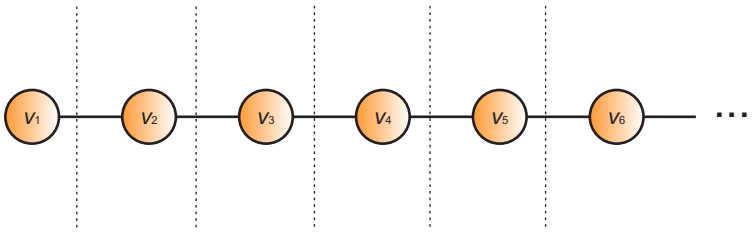


Figura 3.6: Grafo G' con $m = n - 1$ aristas dispuestas a modo de *path* y $CW = 1$.

Si se tuviera que $m = n$, sería necesario añadir una arista extra al *path* representado en la Figura 3.6, lo que necesariamente haría incrementar el *cutwidth* de G' a 2, obteniendo $CW(G') = 2 \leq CW_f(G)$ para cualquier ordenación de los vértices de G.

La tercera y última posibilidad (que se corresponde con el caso más común) es que G tenga un mayor número de aristas que de vértices ($m > n$). En este caso, se ha definido un procedimiento para distribuir las aristas y los vértices en G', de manera que se minimice el *cutwidth* de dicho grafo auxiliar.

El procedimiento de distribución de vértices y aristas comienza disponiendo todos los vértices del grafo en una línea recta. A continuación, se colocan las aristas que unen a vértices consecutivos en la ordenación (entre v_i y v_{i+1} para todo i) tal y como se muestra en la Figura 3.6. Estas aristas pueden considerarse como aristas de longitud 1, pudiendo colocar un total de $n - 1$ aristas de esta longitud. Como se ha descrito anteriormente, si sólo se colocasen dichas aristas, se obtendría un $CW_f(G') = 1$. Una vez colocadas todas las posibles aristas de longitud 1, y dando por hecho que se dispone de más aristas por colocar, el *cutwidth* se incrementará en una unidad en el momento en el que se añada una arista nueva. El objetivo será, por lo tanto, colocar el mayor número de aristas de modo que el *cutwidth* sólo aumente dicha unidad. Dado que no se pueden colocar aristas de longitud 1, se pasará a colocar aristas de longitud 2. En concreto, es posible colocar $\lfloor \frac{(n-1)}{2} \rfloor$ aristas de longitud 2 (situadas entre v_i y v_{i+2}) de modo que el *cutwidth* de G' sea igual a 2, tal y como se muestra en la Figura 3.7.

Se puede deducir que el *cutwidth* de un grafo G con n vértices y m aristas, tal que $n \leq m \leq n - 1 + \lfloor \frac{(n-1)}{2} \rfloor$ satisface $CW(G') = 2 \leq CW_f(G)$ para cualquier etiquetado f de los vértices de G. Destacar que cualquier arista extra incrementaría el *cutwidth* a 3.

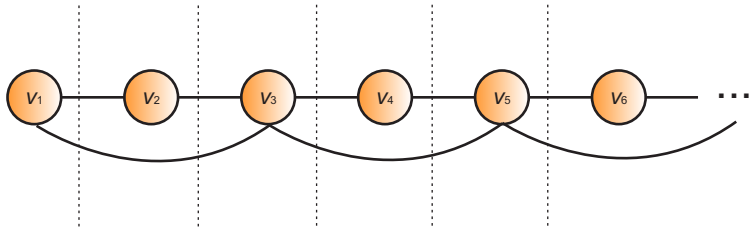


Figura 3.7: Grafo G' con aristas de longitud 1, aristas de longitud 2 y $CW = 2$.

En la Figura 3.8 se muestra cómo añadir $\lfloor \frac{(n-2)}{2} \rfloor$ aristas al grafo de la Figura 3.7 manteniendo el *cutwidth* de G' en 3. Siguiendo el mismo argumento descrito más arriba, el *cutwidth* de un grafo G con n vértices y m aristas con $(n-1) + \lfloor \frac{(n-1)}{2} \rfloor < m \leq (n-1) + \lfloor \frac{(n-1)}{2} \rfloor + \lfloor \frac{(n-2)}{2} \rfloor$ satisface que $CW(G') = 3 \leq CW_f(G)$ para cualquier etiquetado f de sus vértices.

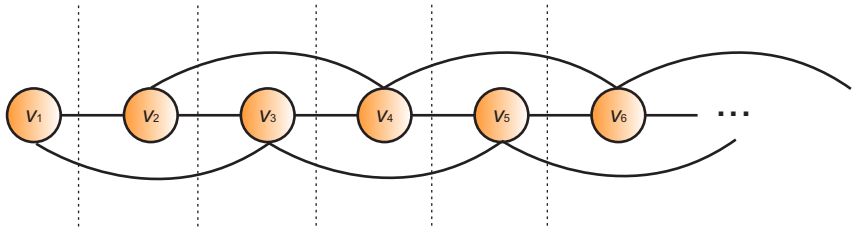


Figura 3.8: Grafo G' con aristas de longitud 1, aristas de longitud 2 y $CW = 3$.

Generalizando esta construcción incremental de G', se observa que existe un máximo de $n-k$ aristas de longitud k (entre v_i y v_{i+k} para todo i) que pueden ser añadidas a G'. Las primeras $\lfloor \frac{(n-1)}{k} \rfloor$ aristas incrementarían el *cutwidth* de G' en una unidad; las siguientes $\lfloor \frac{(n-2)}{k} \rfloor$ en otra unidad; las siguientes $\lfloor \frac{(n-3)}{k} \rfloor$ en una unidad más y así sucesivamente hasta que las $n-k$ aristas de longitud k hubieran sido añadidas. Tras añadir todas las aristas de longitud k, el *cutwidth* de G' (al que previamente habrían sido añadidas todas las aristas con longitud l, con $l = 1$ hasta $l = k-1$) se hubiera visto incrementado en k unidades. De manera genérica se puede calcular el valor de LB_5 mediante la siguiente expresión recursiva:

$$LB_5(M, l, i) = \begin{cases} 0 & \text{si } M \leq 0 \\ 1 + LB_5(M - \lfloor \frac{|U|-i}{l} \rfloor, l, i+1) & \text{si } (M > 0) \wedge (i < l) \\ 1 + LB_5(M - \lfloor \frac{|U|-i}{l} \rfloor, l+1, 1) & \text{si } (M > 0) \wedge (i = l) \end{cases}$$

donde M representa al número de aristas aún sin asignar, U es el conjunto de vértices no etiquetados, $l = \mid f(v_i) - f(v_j) \mid$ representa la longitud de las aristas, para cualquier arista (v_i, v_j) e i representa la etiqueta del vértice en el que se comienza a colocar aristas. El número máximo de aristas con longitud l que se puede colocar en cada llamada recursiva (incrementando el *cutwidth* en una unidad) viene determinado por $\lfloor \frac{|U|-i}{l} \rfloor$. Dado un grafo $G(V, E)$ la llamada

inicial a la expresión recursiva para calcular esta cota sería $LB_5(|E|, 1, 1)$.

El *cutwidth* de G' ofrece una cota inferior del *cutwidth* de cualquier grafo G que tenga el mismo número de vértices y aristas, con independencia de cuál sea la estructura del mismo.

3.3.3. Recorridos del árbol de exploración

El recorrido de un árbol de exploración se lleva a cabo empleando una estrategia de ramificación de los nodos del mismo, con el objetivo de encontrar una solución óptima al problema representado. Todo recorrido comienza en la raíz del árbol, a partir de la cual, en cada iteración del proceso de exploración, se define cuál será el siguiente nodo que debe ser visitado. Los nodos candidatos a ser el siguiente en visitar serán nodos hijos de algún nodo previamente visitado. El recorrido finaliza cuando todos los nodos del árbol han sido explorados o descartados.

La estrategia de ramificación está basada en criterios que permiten asignar una prioridad determinada a cada nodo candidato a ser explorado. Asignar prioridades tiene como objetivo principal guiar la búsqueda hacia soluciones óptimas. La prioridad de los nodos del árbol puede establecerse empleando criterios relacionados con la posición de ese nodo dentro del árbol de exploración, o bien con la calidad de la solución que representa (ya sea real o estimada).

Estrategias de ramificación basadas en la posición del nodo en el árbol de exploración

Dos estrategias de ramificación bien conocidas en las que el criterio de ramificación se basa en la posición de los nodos dentro del árbol de exploración son: la estrategia «en profundidad primero» (DFS, del inglés *Depth First Search*) y la estrategia «en anchura primero» (BFS, del inglés *Breadth First Search*). A continuación, se describe cada una de ellas de manera detallada:

- **DFS**: la estrategia *Depth First Search* o «en profundidad primero», es una estrategia algorítmica diseñada para recorrer los nodos de un árbol en un orden específico. Está basada en la asignación de mayor prioridad a los nodos candidatos que se encuentran más profundos en el árbol de exploración, es decir, el objetivo será alcanzar las hojas del árbol lo antes posible. Para ello, cada vez que se explora un nodo del árbol, antes de explorar los hermanos de éste, se explorarán sus propios hijos y así sucesivamente hasta llegar a un nodo hoja. Cuando se alcanza un nodo hoja, se pasa a explorar un nodo hermano del nodo más profundo explorado, que tenga aún hermanos sin explorar.

  En la Figura 3.9 se muestran los nodos de un árbol de exploración y un número indicando la iteración en la que serían explorados por un algoritmo que siguiera la estrategia DFS. Dicho recorrido comienza por el nodo que ocupa la raíz del árbol y recorre el camino más corto hasta alcanzar una hoja del mismo. A continuación, retrocede en el árbol el mínimo

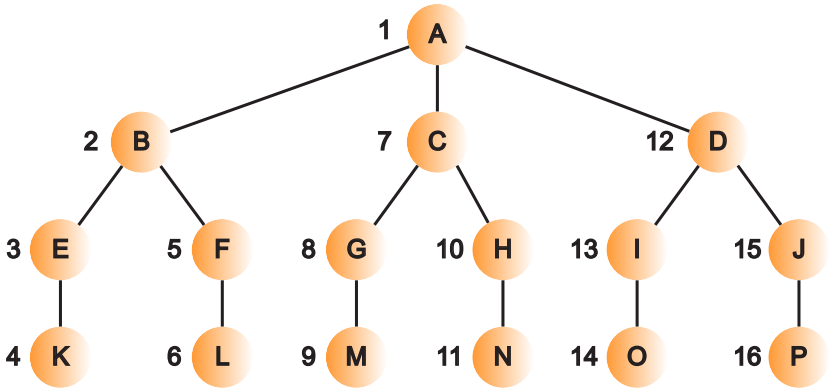


Figura 3.9: Recorrido DFS de un árbol de exploración.

número de niveles posible, hasta que es capaz de encontrar otra rama que conduzca a una hoja no explorada.

El bien conocido algoritmo de «*Backtracking*» utiliza este tipo de recorrido del árbol de exploración en el proceso de búsqueda de la solución óptima.

- **BFS**: la estrategia *Breadth First Search* o «en anchura primero», también es una estrategia algorítmica diseñada para recorrer todos los nodos de un árbol en un orden específico. Está basada en la asignación de mayor prioridad a los nodos candidatos que se encuentran menos profundos en el árbol de exploración, es decir, el objetivo será recorrer los nodos del árbol que se encuentran en niveles superiores lo antes posible. Para ello, cada vez que se explora un nodo del árbol, antes de explorar los hijos de éste, se explorarán todos sus hermanos hasta que no quede ninguno por explorar. En ese momento se explorará un nodo hijo, aún sin explorar, de alguno de los nodos ya explorados.

  En la Figura 3.10 se muestran los nodos de un árbol de exploración y un número indicando la iteración en la que serían explorados por un algoritmo que siguiera la estrategia BFS.

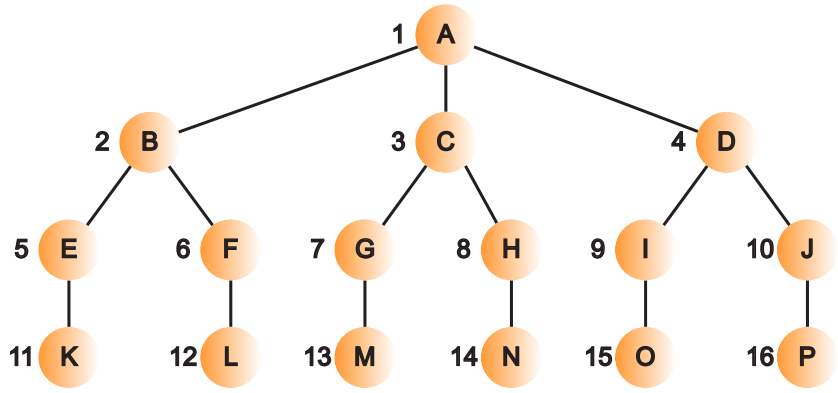


Figura 3.10: Recorrido BFS de un árbol de exploración.

Estrategias de ramificación basadas en la calidad de la solución

El otro tipo de estrategias de ramificación anteriormente mencionadas, son aquéllas basadas en criterios relacionados con la calidad de la solución que contiene cada nodo. Estas estrategias podrían ser clasificadas en dos grandes grupos: real y estimada. En ambos grupos, en cada iteración, se escoge el nodo más prometedor, es decir, aquél que se espera alcance una solución completa de mejor calidad. Dicho nodo será el que tenga un menor valor de la función objetivo (real o estimado) si el problema es de minimización, o el que tenga un mayor valor, si el problema es de maximización. Es importante recordar que los nodos candidatos a ser el siguiente nodo en explorar, son siempre nodos hijos de algún nodo ya explorado. A continuación, se repasan detalladamente cada una de las dos estrategias mencionadas:

- **Valor real de la función objetivo:** a excepción de la raíz, para el resto de nodos del árbol de exploración se puede calcular el valor de la función objetivo de la solución que representa dicho nodo. Para ello, se evaluará la parte de la solución que esté construida en ese nodo, teniendo en cuenta únicamente los vértices y aristas que participan en la solución parcial. Dicho valor certifica que el valor de la función objetivo de los nodos hijos del nodo evaluado será, al menos, como el valor de la función objetivo del nodo que se encuentra en ese punto del árbol de exploración.

- **Valor estimado de la función objetivo:** además de evaluar la solución parcial que está representada en un nodo determinado, también es posible realizar estimaciones que permitan predecir cuál será el valor mínimo (si el problema es de minimización) o máximo (si el problema es de maximización) que tendrán los hijos de dicho nodo. Estas estimaciones están basadas en las denominadas funciones de cota, estudiadas previamente en la Sección 3.3.2. Dichas funciones se centran en algún aspecto de la función objetivo del problema abordado que resulte de utilidad para realizar la estimación.

Cuando se emplean estrategias de ramificación basadas en la calidad de los nodos candidatos, es común emplear una combinación entre las estrategias de valor real y valor estimado de la función objetivo. En la Figura 3.11 se muestra la iteración en la que se recorrerían los nodos de un árbol de exploración, si para cada nodo candidato, se conociera el valor real o estimado de la función objetivo al alcanzar ese nodo. En dicha figura, cada nodo del árbol tiene asociado, junto a su identificador, un valor que representa el valor de la función objetivo si se alcanzara dicho nodo. Por ejemplo, el nodo A, que representa a la raíz del árbol, se recorrería en primer lugar y no tiene asignado un valor de la función objetivo, ya que no es posible determinarlo en este nodo. El nodo B, tiene asignado un valor de la función objetivo de 2 y se recorre en segundo lugar y así sucesivamente hasta alcanzar el nodo O, cuyo valor de la función objetivo es 9 y que se examinaría en último lugar. Junto a cada nodo se especifica un número indicando la iteración

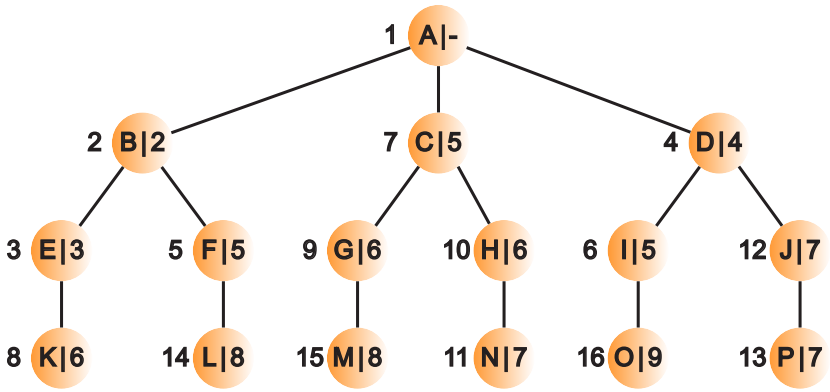


Figura 3.11: Ejemplo de recorrido de un árbol de exploración basado en la calidad de la solución parcial o completa de cada nodo.

en la que se recorrería dicho nodo. En un problema de minimización, cuanto menor sea el valor de la función objetivo de un nodo, mayor prioridad tiene.

En esta Tesis Doctoral se proponen tres estrategias de ramificación diferentes basadas en algunas de las ideas anteriormente descritas. Cada una de estas estrategias, en combinación con las funciones de cota propuestas en la Sección 3.3.2, forman un algoritmo de Ramificación y Acotación para la resolución del CMP. Estos algoritmos han sido denominados empleando el acrónimo de su nombre en inglés «*Branch and Bound*» (BB) como BB1, BB2 y BB3.

Además de las funciones de cota para calcular cotas inferiores al CMP, los algoritmos BB1, BB2 y BB3 comparten el algoritmo para el cálculo de una cota superior inicial. El valor de la solución obtenida por dicho algoritmo, descrito en la Sección 3.4, permite podar ramas poco prometedoras del árbol de exploración desde el primer momento del proceso de búsqueda. Dicha cota superior se irá actualizando durante el proceso de exploración, a medida que se vayan alcanzando soluciones de mayor calidad.

Algoritmo BB1

Está basado en la combinación de la estrategia de ramificación del árbol de exploración DFS, con las cotas propuestas en la Sección 3.3.2. Se trata de un procedimiento de búsqueda recursivo que prioriza la expansión en profundidad del árbol de exploración.

Durante el proceso de búsqueda de la solución óptima, el algoritmo BB1 trata de ramificar el nodo candidato más profundo. En el caso de que exista más de un nodo candidato situado a la misma profundidad máxima, escoge el siguiente nodo a expandir de manera lexicográfica entre ellos.

Durante la ejecución del algoritmo, se almacena el valor de la mejor solución encontrada. Este valor se actualiza cada vez que se alcanza un nodo hoja y, el valor de la función objetivo de la solución asociada a ese nodo, es menor que el mejor valor almacenado hasta ese momento.

A cada paso del algoritmo, cuando se explora un nodo del árbol, se evalúan las funciones de cota descritas en la Sección 3.3.2 y se decide si se deben seguir explorando los hijos del mismo o, por el contrario, se debe de podar esa rama. Cuando el máximo entre las estimaciones proporcionadas por las distintas funciones de cota, es mayor o igual que el valor de la mejor solución encontrada hasta ese momento, no es necesario expandir esa rama, ya que la solución que se alcanzará en cualquiera de sus hijos, será peor o igual que la mejor solución encontrada hasta ese momento. A la acción de evitar que se ramifiquen los hijos de un nodo determinado, se la denomina poda.

El pseudocódigo del BB1 se muestra en Algoritmo 3.1. Este procedimiento recibe como parámetro el siguiente nodo que se desea explorar. Si se trata de un nodo hoja, evalúa la solución que contiene y actualiza la mejor solución encontrada, en caso de que la calidad de ésta sea mejor que la de la mejor solución que se tuviera de anteriores iteraciones. Si no es un nodo hoja, el algoritmo calcula el máximo de las cotas inferiores propuestas. En caso de que el valor máximo de estas cotas sea menor que el valor de la mejor solución encontrada hasta ese momento, se pasa a expandir, lexicográficamente, cada uno de sus hijos con llamadas recursivas. Si por el contrario, el valor de la estimación es mayor o igual que la mejor solución encontrada hasta el momento, se poda dicha rama y no se explora ninguno de sus hijos. El algoritmo comienza explorando el nodo que representa la raíz del árbol, con la llamada BB1($nodo_k$), siendo $nodo_k = \{S = \varnothing;\ g(u) = 0,\ \forall\, u \in V\}$ y $k = 0$.

Algoritmo 3.1 Algoritmo de Ramificación y Acotación BB1

```
1:  procedimiento BB1(nodo_k)
2:  UB ← valor de la F.O. de una solución generada con un heurístico;
3:  Sea (S, g) la solución parcial asociada a nodo_k, siendo k la última etiqueta asignada;
4:  si nodo_k es un nodo hoja entonces
5:      Calcular CW_f(G) como el valor de la solución asociada al nodo;
6:      si CW_f(G) < UB entonces
7:          UB = CW_f(G);
8:      fin si;
9:  si no
10:     Calcular LB = max {LB_1, LB_2, LB_3, LB_4, LB_5};
11:     si LB < UB entonces
12:         Sea U el conjunto de vértices no etiquetados;
13:         k = k + 1;
14:         mientras U ≠ ∅ hacer
15:             Seleccionar u ∈ U en orden lexicográfico;
16:             U = U \ {u};
17:             Establecer nodo_k = {S = S ∪ {u}; g(u) = k} ;
18:             BB1(nodo_k);
19:         fin mientras;
20:     fin si;
21: fin si;
22: fin BB1;
```

Este algoritmo presenta la ventaja de que es capaz de alcanzar nodos hojas con gran rapidez, lo que puede ayudar a reducir la cota superior del problema, que a su vez es empleada para podar ramas del árbol de exploración. Además, requiere almacenar muy pocos nodos del árbol de exploración en memoria principal. Su principal desventaja reside en que, en el caso de que la exploración del algoritmo esté limitada por tiempo y el algoritmo no haya sido capaz de explorar todo el árbol, no permite determinar una buena cota inferior.

Algoritmo BB2

El algoritmo de Ramificación y Acotación BB2 puede considerarse una variante de BB1. Este algoritmo también combina una estrategia de ramificación en profundidad del árbol de exploración, con las cotas propuestas en la Sección 3.3.2. No obstante, introduce una pequeña variación en la selección del siguiente nodo a explorar, cuando existe más de un candidato situado a profundidad máxima. BB1 selecciona el nodo lexicográficamente, mientras BB2 evalúa la cota estimada de cada nodo candidato, ordenándolos según este valor y comenzando por aquél que tenga una menor cota. El pseudocódigo de este algoritmo se muestra en Algoritmo 3.2.

Algoritmo 3.2 Algoritmo de Ramificación y Acotación BB2

```
1:  procedimiento BB2(nodo_k)
2:  UB ← valor de la F.O. de una solución generada con un heurístico;
3:  Sea (S, g) la solución parcial asociada a nodo_k, siendo k la última etiqueta asignada;
4:  si nodo_k es un nodo hoja entonces
5:      Calcular CW_f(G) como el valor de la solución asociada al nodo;
6:      si CW_f(G) < UB entonces
7:          UB = CW_f(G);
8:      fin si;
9:  si no
10:     Calcular LB = max {LB_1, LB_2, LB_3, LB_4, LB_5};
11:     si LB < UB entonces
12:         Sea U el conjunto de vértices no etiquetados;
13:         Sea PQ una cola de prioridad vacía;
14:         k = k + 1;
15:         mientras U ≠ ∅ hacer
16:             Seleccionar u ∈ U;
17:             U = U \ {u};
18:             Establecer nodo_k = {S = S ∪ {u}; g(u) = k};
19:             Para nodo_k calcular prioridad LB = max {LB_1, LB_2, LB_3, LB_4, LB_5};
20:             Añadir nodo_k a PQ en orden creciente de LB;
21:         fin mientras;
22:         mientras PQ ≠ ∅ hacer
23:             Extraer nodo_k de PQ en orden de prioridad;
24:             BB2(nodo_k);
25:         fin mientras;
26:     fin si;
27: fin si;
28: fin BB2;
```

Al igual que BB1, el algoritmo BB2 comienza explorando el nodo que representa la raíz del árbol, con la llamada BB2($nodo_k$), siendo $nodo_k = \{S = \varnothing;\ g(u) = 0,\ \forall\, u \in V\}$ y $k = 0$. La llamada recursiva se produce después de ordenar, según la prioridad establecida por las funciones de cota, a los nodos que se encuentran a la misma profundidad máxima, es decir, aquellos nodos hijos del nodo que se está explorando.

Esta técnica permite dar prioridad a los nodos más prometedores, permitiendo al algoritmo alcanzar cotas superiores de mejor calidad en menos tiempo que BB1. Además, palía parte de la desventaja en el cálculo de cotas inferiores de BB1, cuando el algoritmo no es capaz de examinar todo el árbol de exploración en el tiempo previsto. Esto es debido a que en el primer nivel del árbol de exploración se ha evaluado la cota de todos los nodos hijos de la raíz. En ese nivel, la cota de menor valor de todos los nodos hijos es una cota inferior al problema.

Algoritmo BB3

El algoritmo de Ramificación y Acotación BB3 es un algoritmo iterativo basado en la expansión del árbol de exploración según la calidad real o estimada de cada nodo candidato. Para evaluar lo prometedor que es un nodo se emplean las distintas funciones de cota propuestas en la Sección 3.3.2. Todo nodo candidato tendrá, por lo tanto, un valor de prioridad asociado, establecido según la evaluación de dichas funciones de cota. Una vez evaluado, el nodo candidato será insertado en una cola de prioridad, que ordena los nodos de más a menos prometedor, según el valor de prioridad asignado.

El pseudocódigo de este algoritmo se muestra en Algoritmo 3.3. Inicialmente la exploración comienza añadiendo a la cola de prioridad la raíz del árbol de exploración. Mientras que la cola de prioridad no se vacíe, se desencola el nodo situado en la cabeza de la misma y se continúa la búsqueda expandiendo los hijos de éste. Una vez extraído un nodo de la cabeza de la cola, se comprueba si el valor de prioridad que tiene asociado aún permite conducir a una solución mejor que la mejor solución encontrada (paso 15 del pseudocódigo) ya que podría darse el caso de que el nodo fuera prometedor cuando se encoló, pero que no lo sea en este momento del recorrido, por haber encontrado una solución mejor. Si el nodo no fuera prometedor, la exploración habría terminado, ya que todos los nodos en la cola son igual o menos prometedores que el nodo actual. Si por el contrario, el nodo explorado es todavía prometedor, se añaden a la cola de prioridad todos sus hijos que sean también prometedores (paso 23 del pseudocódigo) permitiendo así ahorrar espacio, al evitar que se encolen nodos que no conducen a una solución mejor que la actual. El proceso continúa hasta que no queden nodos por explorar en la cola de prioridad.

Este algoritmo presenta la ventaja de que es capaz de proporcionar cotas inferiores de gran calidad cuando no es posible examinar el árbol de exploración por completo, debido a restricciones de tiempo. El valor de dicha cota inferior será el valor de prioridad asociado al

nodo que se encuentre en la cabeza de la cola. En cuanto a las desventajas, cabe destacar el gran consumo de memoria que requiere para almacenar los nodos contenidos en la cola de prioridad. Cuando el tamaño del árbol asociado al grafo que representa el problema de entrada al algoritmo es grande, será necesario volcar parte de la cola de prioridad a memoria secundaria, durante el proceso de exploración. La utilización de memoria secundaria implica, a su vez, un mayor consumo de tiempo. Además, dado que explora los nodos ordenadamente según lo prometedores que son, necesita más tiempo para alcanzar las primeras hojas del árbol de exploración, que los algoritmos que recorren el árbol priorizando la expansión de los nodos más profundos. Esto conduce a que sea más difícil encontrar cotas superiores más ajustadas que permitan podar más rápidamente otras ramas del árbol de exploración. No obstante, este problema puede ser minimizado si se dispone de un algoritmo heurístico capaz de construir una solución inicial de muy alta calidad.

Algoritmo 3.3 Algoritmo de Ramificación y Acotación BB3

```
1: procedimiento BB3()
2: UB ← valor de la F.O. de una solución generada con un heurístico;
3: k = 0;
4: Establecer nodo_k = {S = ∅; g(u) = 0; ∀u ∈ V};
5: Sea PQ una cola de prioridad vacía;
6: Añadir nodo_k a PQ ordenado por su prioridad;
7: mientras PQ ≠ ∅ hacer
8:     nodo_k = extraer nodo de PQ en orden de prioridad;
9:     si nodo_k es un nodo hoja entonces
10:        Calcular CW_f(G) como el valor de la solución asociada al nodo;
11:        si CW_f(G) < UB entonces
12:            UB = CW_f(G);
13:        fin si;
14:    si no
15:        si LB < UB entonces
16:            Sea U el conjunto de vértices no etiquetados;
17:            Sea k la última etiqueta asignada en el nodo actual, nodo_k;
18:            mientras U ≠ ∅ hacer
19:                Seleccionar u de U en orden lexicográfico;
20:                U = U \ {u};
21:                Establecer nodo_{k+1} = {S = S ∪ {u}; g(u) = k + 1};
22:                Para nodo_{k+1} calcular LB = max {LB_1, LB_2, LB_3, LB_4, LB_5};
23:                si LB < UB entonces
24:                    Añadir nodo_{k+1} a PQ ordenado por LB;
25:                fin si;
26:            fin mientras;
27:        fin si;
28:    fin si
29: fin mientras;
30: fin BB3;
```

3.3.4. Almacenamiento del árbol de exploración en memoria

En el diseño de algoritmos basados en árboles de exploración para la búsqueda de soluciones exactas, es esencial definir estructuras de datos eficientes para el almacenamiento de los nodos del árbol. Dado que éste se recorre de manera ordenada, empleando para ello una serie de criterios, es común utilizar estructuras de datos tales como pilas, colas, colas de prioridad o montículos, entre otras, para almacenar los nodos del árbol [42].

Recorrer un árbol de exploración requiere mantener información del mismo en la memoria principal del ordenador. Según el tamaño del grafo de entrada al algoritmo, las necesidades de memoria pueden llegar a exceder la capacidad de la memoria principal, tal y como se pudo comprobar en la Sección 3.3.1. Generar los nodos del árbol de manera dinámica a medida que se van explorando resulta de utilidad para reducir la cantidad de información almacenada en memoria principal, de manera simultánea.

En relación con el recorrido de los nodos de un árbol de exploración, éstos se podrían clasificar en cuatro categorías diferentes: nodos visitados, nodos candidatos, nodos por explorar y nodos descartados. Los nodos visitados son aquéllos que ya han sido evaluados en busca de la solución óptima al problema y cuyos hijos son, o bien candidatos, o bien visitados, o bien descartados. Los nodos candidatos son el subconjunto de nodos del árbol que tienen posibilidades de ser el siguiente nodo a examinar, debido a que son hijos de algún nodo ya visitado y continúan siendo nodos prometedores. Los nodos por explorar son aquéllos que no han sido visitados aún, ni tampoco son candidatos a ser visitados en la siguiente iteración del algoritmo, ya que su padre no ha sido visitado todavía. Por último, los nodos descartados son aquéllos que han sido podados, dado que la calidad de la solución estimada por alguna función de cota ha determinado que no conducirán a una solución mejor que la solución de la que se dispone.

Los nodos descartados, los nodos ya visitados y los nodos por explorar no necesitan estar en memoria principal en cada iteración del algoritmo. Por lo tanto, en cada momento, los únicos nodos que deben estar almacenados en la memoria principal serán los nodos candidatos y el nodo que se esté examinando en ese instante. Una vez se examina un nodo, sus nodos hijos, si los tuviera, pueden pasar a ser candidatos, por lo que habría que generarlos y almacenarlos en la estructura de datos correspondiente (siempre y cuando fueran prometedores) o bien descartarlos.

Los dos recorridos del árbol de exploración anteriormente mencionados (DFS y BFS) pueden ser implementados utilizando estructuras de datos tipo pila y cola respectivamente.

En la Figura 3.12.a se muestra un ejemplo del recorrido de un árbol de exploración para un grafo de tres vértices, empleando la estrategia DFS. Junto a cada nodo se indica, mediante un número, el número de la iteración en la que dicho nodo sería explorado. En la Figura 3.12.b se representa una estructura de datos tipo pila para almacenar los nodos generados durante ese recorrido. En dicha figura, cada fila representa una iteración del algoritmo, siendo necesarias

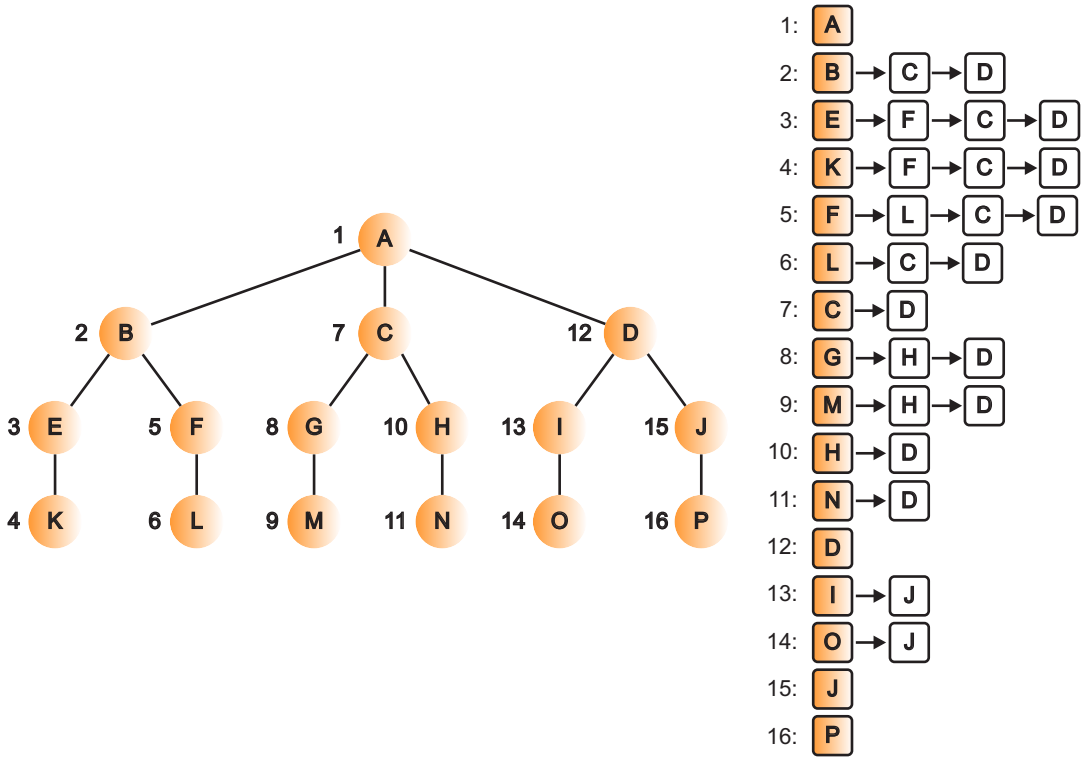
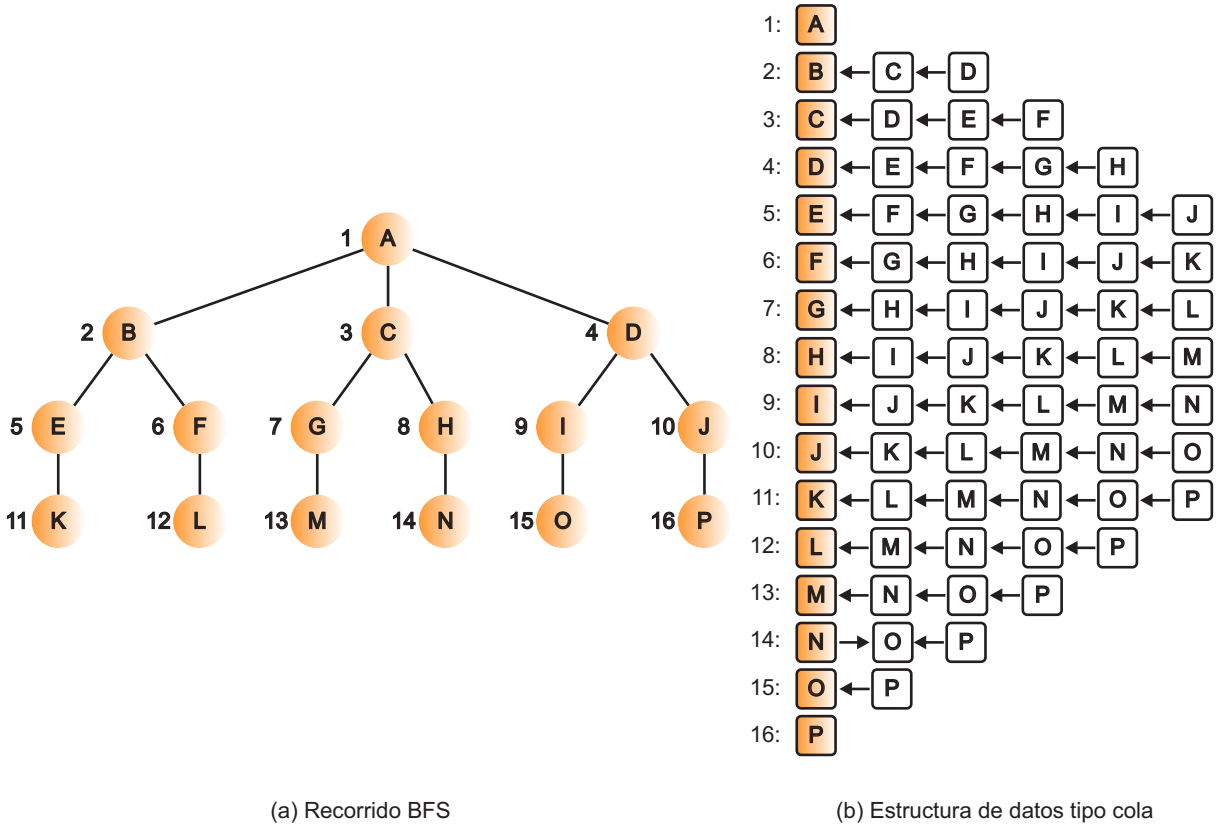


Figura 3.12: Ejemplo de recorrido DFS de un árbol de exploración, empleando una estructura de datos tipo pila.

un total de 16 para recorrer todos los nodos del árbol. En naranja se muestra el nodo que es examinado en cada iteración y que por lo tanto debe ser desapilado cara a la siguiente iteración. En blanco, se muestran los nodos que forman parte de la pila (nodos candidatos) pero que no serán el nodo elegido a examinar en esa iteración. Una vez que se examina un nodo, se añaden los hijos prometedores de éste a la pila. En una estructura de datos tipo pila, los nodos se añaden por a la cima de la misma, de modo que los últimos nodos en ser añadidos serán los primeros en ser examinados. El proceso finaliza cuando no quedan nodos en la pila.

Análogamente, en la Figura 3.13.a se muestra un ejemplo del recorrido de un árbol de exploración empleando la estrategia BFS. En la Figura 3.13.b se representa una estructura de datos tipo cola para almacenar los nodos generados durante este recorrido. La diferencia fundamental con la descripción de la Figura 3.12.b reside en que los nuevos nodos que se añaden a la estructura de datos, se añaden al final de la cola y no serán examinados hasta que todos los nodos que ya estaban en la cola anteriormente hayan sido examinados.

Como se puede observar al comparar la Figura 3.12.b con la Figura 3.13.b, es necesario mantener muchos más nodos en memoria principal cuando se realizan recorridos en anchura (como BFS) que cuando se realizan recorridos en profundidad (como DFS).



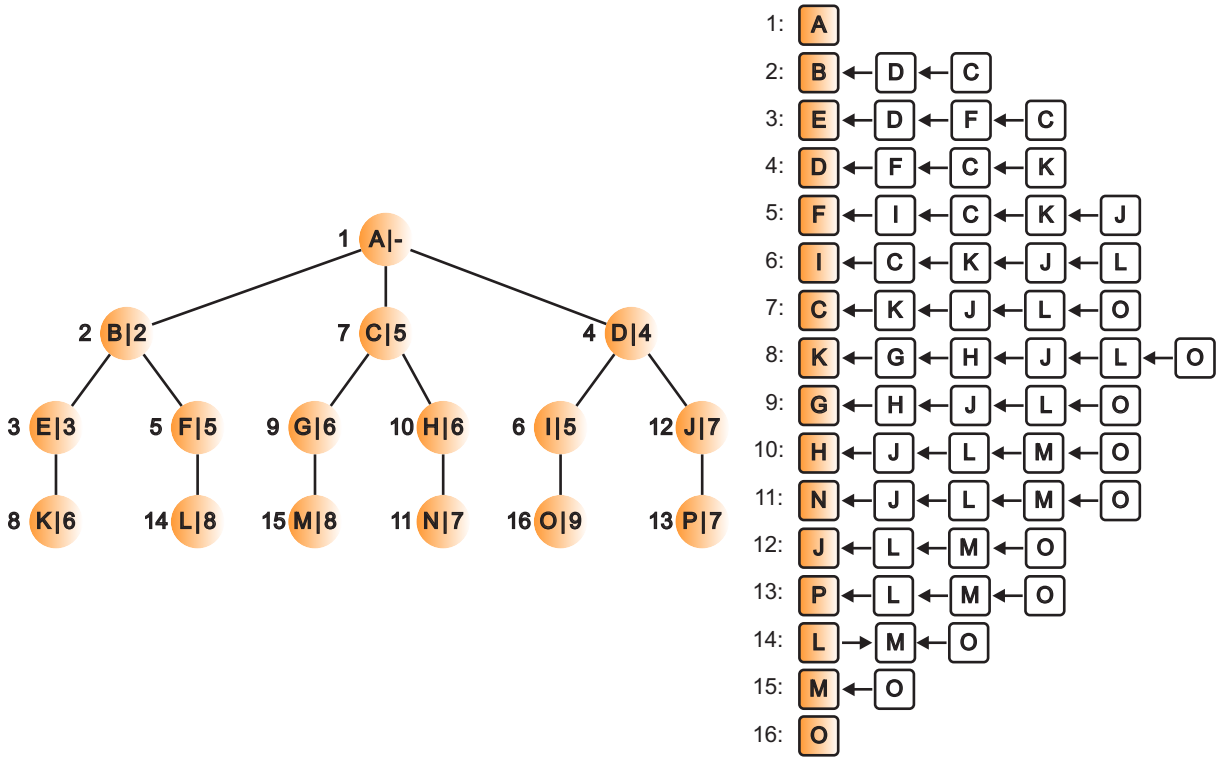
(a) Recorrido BFS (b) Estructura de datos tipo cola

Figura 3.13: Ejemplo de recorrido BFS de un árbol de exploración, empleando una estructura de datos tipo cola.

Cola de prioridad

Cuando se tiene en cuenta el valor de la función objetivo o el valor estimado por una cota a la hora de decidir cuál será el siguiente nodo del árbol de exploración a examinar, es necesario emplear una estructura de datos más compleja. Esta estructura debe permitir almacenar los nodos del árbol de manera ordenada, basándose en una prioridad. En concreto, es común utilizar la estructura de datos cola de prioridad para tal fin. En las colas de prioridad, todo elemento tiene una prioridad asociada que permite ordenar los elementos de la misma. Dicha prioridad puede ser calculada evaluando la función objetivo parcialmente construida o bien realizando una estimación de lo buena que puede llegar a ser la solución, empleando para ello una función de cota.

En la Figura 3.14.a se muestra un ejemplo de recorrido de un árbol de exploración, basado en el valor de la función objetivo o de la cota de los nodos que lo componen. En dicha figura, cada nodo tiene asociado, además de su identificador, un número que indica el valor que alcanzaría la función objetivo en dicho nodo. Fuera del nodo, se incluye otro número que identifica la iteración en la que dicho nodo será explorado. Por otro lado, en la Figura 3.14.b se muestra una estructura de datos tipo cola de prioridad, que permite almacenar los nodos generados durante el recorrido



(a) Recorrido basado en la prioridad de cada nodo

(b) Estructura de datos tipo cola de prioridad

Figura 3.14: Ejemplo de recorrido de un árbol de exploración basado en la prioridad de sus nodos, empleando una estructura de datos tipo cola de prioridad.

mostrado en la Figura 3.14.a y determinar, en cada iteración del algoritmo, cuál será el siguiente nodo a examinar. Cada fila de esta figura, identificada con un número del 1 al 16 representa una iteración del proceso de exploración y, por lo tanto, el conjunto de nodos almacenados en la cola de prioridad en ese instante. En naranja se muestran los nodos que son explorados en la iteración y, en blanco, el resto de nodos candidatos almacenados en la cola.

Inicialmente el recorrido parte de la raíz del árbol de exploración, por lo que se encola dicho nodo en la cola de prioridad. Al examinar el nodo que contiene la raíz, éste se desencola y se encolan los nodos hijos del nodo examinado, en el orden indicado por el valor de la solución parcial (o de la estimación) de cada uno. En la siguiente iteración del algoritmo se desencola el nodo con mayor prioridad y se encolan sus hijos, en la posición que les corresponda según la calidad que atesoren. El algoritmo acaba cuando no quedan más nodos en la cola de prioridad.

Almacenamiento de nodos del árbol de exploración en memoria secundaria

Como se ha indicado anteriormente, generar el árbol de exploración de manera dinámica evita mantener en memoria principal nodos visitados previamente y nodos que no van a ser explorados en las siguientes iteraciones del algoritmo. No obstante, en ocasiones, el número de nodos candidatos a ser el siguiente en ser visitado es tan elevado que es imposible mantenerlos

todos en la memoria principal del ordenador. Este suceso depende fundamentalmente del tamaño del grafo y del tipo de recorrido del árbol de exploración que se realice.

En recorridos en profundidad, el número de nodos que se mantienen de manera simultánea en memoria principal asciende, como máximo, a un nodo por cada nivel del árbol de exploración. Este tipo de algoritmos son implementados habitualmente de manera recursiva y utilizando sólo memoria principal, ya que son eficientes desde el punto de vista del consumo de memoria. En cambio, en el caso de recorridos en anchura, el número de nodos que pueden llegar a ser candidatos de manera simultánea, y que por lo tanto estarían en memoria principal, podría llegar a ser $n!$.

Cuando la prioridad que define el recorrido del árbol está basada en la calidad de la solución de cada nodo candidato, el recorrido se asemeja a un recorrido en anchura. En general, para problemas de minimización, los nodos situados en niveles superiores del árbol (más cercanos a la raíz) tienen un mejor valor de la función objetivo que los nodos que se encuentran en niveles inferiores. Esto es debido a que la solución incompleta contenida en ese nodo, tiene menos elementos asignados, por lo tanto la cota estimada es más baja. Este comportamiento conduce a explorar antes los nodos de niveles superiores, lo que implica mantener un gran número de nodos en memoria de manera simultánea, a medida que se desciende en el árbol de exploración.

Si el número de nodos candidatos es tan grande que no es posible conservarlos todos en memoria principal, es necesario volcar parte de ellos a memoria secundaria. En este caso, es imprescindible mantener un registro preciso de qué nodos hay en memoria secundaria, para poder recuperarlos cuando deban ser explorados según su prioridad.

De los recorridos del árbol de exploración propuestos en la Sección 3.3.3, únicamente BB3 necesita utilizar el volcado a memoria secundaria durante su ejecución. Este recorrido está basado en la utilización de una cola de prioridad para organizar los nodos candidatos, de acuerdo con la calidad de la solución parcial que contienen o del valor de la cota para dicho nodo. Este hecho permite realizar un volcado inteligente de parte de los nodos del árbol. Algunos de los aspectos a decidir durante el proceso de volcado son los siguientes:

- **Qué nodos volcar**: es necesario decidir qué nodos de la cola de prioridad deben ser volcados a memoria secundaria. A priori, lo más ventajoso es volcar los nodos con una prioridad menor, es decir, aquéllos que contengan una solución cuya función objetivo o cota tiene un valor mayor, para un problema de minimización. El motivo es que es posible que nunca lleguen a ser visitados y, por lo tanto, no sería necesario recuperarlos de nuevo. Esto sucede cuando alguno de los nodos más prometedores, mantenidos en memoria principal, alcanza una solución mejor que la contenida o estimada en ese nodo candidato.

- **Cuántos nodos volcar**: el número de nodos a volcar debe ser variable según el tamaño de la instancia, dado que este tamaño determina, a su vez, el tamaño de los nodos. Esto

es debido a que los nodos deben almacenar, entre otra información, una solución para dicha instancia. A su vez, esto implica que el número de nodos a volcar deba decidirse dinámicamente según un porcentaje respecto al número de nodos en memoria en un momento dado. Esta decisión se tomará cuando se haya superado el límite de consumo de memoria establecido para que el algoritmo continúe su ejecución de manera eficiente.

- **Cuándo volcarlos**: en función del tamaño del problema puede ser necesario realizar un volcado de nodos del árbol de exploración a memoria secundaria. Por lo tanto, se debe tener un mecanismo flexible que permita decidir dinámicamente en qué condiciones se desea volcar información a memoria secundaria. En este caso, lo esencial es que el ordenador disponga de memoria suficiente para examinar el árbol de exploración y, eventualmente, para cargar un bloque de información de disco. Por ello, se ha incluido un parámetro que permite determinar el porcentaje de utilización de la memoria principal a partir del cual se comienza a volcar nodos a memoria secundaria.

- **Cómo volcarlos**: quizá éste sea uno de los aspectos más delicados del proceso de volcado. Una vez definidos qué nodos volcar y en qué momento volcarlos, es importante decidir cómo almacenarlos en disco. Para ello hay que tener en cuenta aspectos tales como la velocidad de acceso (tanto del proceso de escritura como del proceso de lectura) o la gestión de la información en memoria secundaria. Entre las alternativas existentes se ha valorado la utilización de una base de datos y la *serialización*[9] de objetos. Las bases de datos proporcionan mecanismos muy potentes para el almacenamiento y acceso a la información, simplificando notablemente todo lo relacionado con la gestión de la misma. No obstante, el tiempo empleado para almacenar la información es muy superior al necesario para *serializar* objetos.

Algunos de los aspectos anteriores están expresados en el algoritmo BB3 a modo de parámetros, que pueden ser modificados de acuerdo con las características técnicas del ordenador en el que se ejecute el algoritmo.

La configuración del algoritmo BB3 establece el porcentaje de volcado de nodos a memoria secundaria en un 20 % del tamaño de la cola de prioridad. El porcentaje de uso de la memoria principal, a partir de la cual comienza el volcado, está establecido en el 50 % de la memoria del ordenador. Por último, la técnica elegida para volcar los nodos del árbol de exploración a disco es la *serialización* de objetos. En concreto se *serializan* objetos tipo `PriorityQueue`, que contienen el porcentaje establecido de nodos menos prometedores de la cola de prioridad.

Por ejemplo, suponiendo que el algoritmo se está ejecutando en un ordenador con 6 GB de memoria RAM, el volcado a disco se llevará a cabo cada vez que se superen los 3 GB de RAM de

[9] El proceso de *serialización* de objetos consiste en traducir un objeto almacenado en memoria principal a un *array* de *bytes* que pueda ser almacenado en memoria secundaria. Este mecanismo presenta el inconveniente de que es necesario gestionar dónde se almacena cada objeto, qué identificador tiene y qué información contiene.

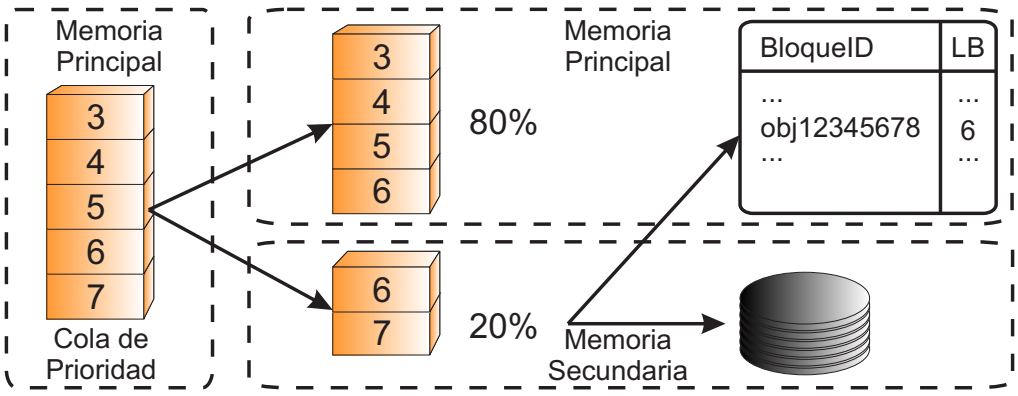


Figura 3.15: Volcado de nodos del árbol de exploración a memoria secundaria.

utilización de memoria. Supóngase también que en un instante determinado, al superar los 3 GB de memoria utilizada hubiera almacenados en la cola de prioridad 10^{10} nodos. En este caso se volcarían a disco los 10^2 nodos menos prometedores y se liberaría dicha memoria, manteniendo la memoria principal por debajo de 3 GB con los 10^8 nodos restantes.

En la Figura 3.15 se representa una cola de prioridad que almacena, originalmente en memoria principal, los nodos del árbol de exploración para el CMP. El valor de la función objetivo o de la cota, de los nodos en dicha cola oscila entre 3 y 7. Cada bloque de la cola representa al conjunto de nodos candidatos que tienen ese valor de prioridad asignado. Los nodos con mayor prioridad en dicha figura, son aquéllos con valor de prioridad 3 (se trata de un problema de minimización) y los de menor prioridad, aquéllos con valor de prioridad 7. Suponiendo que se ha alcanzado el uso del 50 % de la memoria del ordenador donde se ejecuta el algoritmo, se separa el 20 % de los nodos menos prometedores para almacenarlos en memoria secundaria. El proceso comienza dividiendo la cola de prioridad en dos colas, manteniendo una de ellas en memoria principal, con el 80 % de los nodos más prometedores y *serializando* el objeto que representa a la otra, para almacenarlo en disco. En una estructura de datos auxiliar, en memoria principal, se mantiene el identificador del archivo que contiene el objeto en disco (*BloqueID*) y el valor del nodo con una cota mejor (*LB*). En la Figura 3.15 se muestra un ejemplo de identificador (obj12345678) y la cota de la mejor solución en dicho bloque (6). Es importante notar que, dado que se vuelca a memoria secundaria un porcentaje de los nodos vivos del árbol, es posible que haya nodos con la misma prioridad en disco y en memoria principal. Cuando todos los nodos con prioridad 3, 4 y 5 hayan sido explorados, se pasa a explorar los nodos con prioridad 6 comenzando por los nodos que se encuentran en memoria principal y recuperando, a continuación, uno a uno, los bloques de información de disco que contengan nodos con dicha prioridad. El proceso finaliza cuando no queden bloques por explorar ni en memoria principal, ni en memoria secundaria o cuando el valor de la mejor cota sea mayor o igual al valor de la mejor solución encontrada.

3.4. Cota superior inicial

Los algoritmos de Ramificación y Acotación suelen emplear una cota superior inicial que permite acelerar el proceso de poda de ramas poco prometedoras, sin necesidad de tener que alcanzar las hojas del árbol de exploración para ello. Como se ha indicado anteriormente, además de las funciones de cota presentadas en la Sección 3.3.2, los tres algoritmos de Ramificación y Acotación propuestos en esta Tesis Doctoral (BB1, BB2 y BB3) comparten el algoritmo para el cálculo de dicha cota superior inicial. La poda de ramas poco prometedoras del árbol se produce cuando el nodo evaluado tiene un valor estimado mayor o igual que la mejor cota superior encontrada. Ésta, se irá actualizando durante el proceso de exploración, a medida que se alcancen nodos hojas con soluciones de mejor calidad.

El cálculo de una cota superior inicial puede realizarse construyendo soluciones factibles de manera aleatoria y evaluando su función objetivo. No obstante, existen técnicas para obtener cotas superiores de mejor calidad. En concreto, las metaheurísticas han sido utilizadas para generar buenas cotas superiores (cuando la función objetivo consiste en minimizar un valor) para problemas de optimización [172]. Partir de una buena cota superior ayuda a mejorar el rendimiento de los algoritmos de Ramificación y Acotación.

En esta Tesis Doctoral se propone un algoritmo heurístico para generar soluciones al CMP, que permita establecer cotas superiores iniciales de buena calidad. Este algoritmo, que supuso una primera versión del algoritmo heurístico propuesto para la resolución del CMP, detallado en el Capítulo 4, está basado en la metodología GRASP [56]. Cada iteración del procedimiento GRASP incluye la construcción y la mejora de una solución. Este procedimiento se repite un número determinado de iteraciones. Tanto el método constructivo como el de mejora se detallan en las secciones 3.4.1 y 3.4.2 respectivamente.

3.4.1. Método constructivo

El método constructivo propuesto comienza con la creación de una lista de vértices no etiquetados U. Inicialmente $U = V$, siendo V el conjunto de vértices del grafo. En el primer paso del algoritmo, el procedimiento constructivo escoge, de manera aleatoria, un vértice $v \in U$ y le asigna la etiqueta 1. En las sucesivas iteraciones, el algoritmo construye una lista de candidatos (CL, del inglés *Candidate List*) que incluye todos los vértices de U con al menos un vértice adyacente etiquetado. Para cada vértice $u \in CL$ se evalúa la diferencia ($d(u)$) entre el número de vértices adyacentes a u ya etiquetados ($N_L(u)$) y el número de vértices adyacentes a u no etiquetados ($N_U(u)$) de la siguiente manera:

$$d(u) =\mid N_L(u) \mid - \mid N_U(u) \mid$$

En este paso del algoritmo, un procedimiento voraz seleccionaría aquel vértice u^* con un valor $d(u)$ mayor. En cambio, la metodología GRASP construye una Lista de Candidatos Restringida (RCL, del inglés *Restricted Candidate List*) con un porcentaje de los mejores candidatos y selecciona uno de ellos de manera aleatoria. En este caso, $RCL = \{v \in CL \,/\, d(v) \geq th\}$ donde el parámetro th es un umbral que debe ser establecido empíricamente.

El pseudocódigo de este método constructivo se muestra en Algoritmo 3.4.

Algoritmo 3.4 Algoritmo constructivo para la creación de una cota superior inicial

1: **procedimiento** ConstructivoUB()
2: Sean L y U los conjuntos de vértices etiquetados y no etiquetados de un grafo;
3: Inicialmente $L = \varnothing$ y $U = V$;
4: Seleccionar un vértice u de U de manera aleatoria;
5: Asignar la etiqueta $k = 1$ a u;
6: $L = \{u\}$, $U = U \setminus \{u\}$;
7: **mientras** $U \neq \varnothing$ **hacer**
8: $k = k + 1$;
9: Construir $CL = \{v \in U \,/\, (w, v) \in E, \forall\, w \in L\}$;
10: Sean $N_L(v)$ y $N_U(v)$ los conjuntos de vértices adyacentes a v etiquetados y no etiquetados respectivamente;
11: Calcular $d(v) =\mid N_L(v) \mid - \mid N_U(v) \mid \ \forall\, v \in CL$;
12: Construir $RCL = \{v \in CL \,/\, d(v) \geq th\}$;
13: Seleccionar un vértice u de manera aleatoria de la RCL;
14: Etiquetar u con la etiqueta k;
15: $U = U \setminus \{u\}, L = L \cup \{u\}$;
16: **fin mientras**;
17: **fin** ConstructivoUB;

3.4.2. Método de mejora

Una vez construida una solución con el algoritmo constructivo propuesto en la Sección 3.4.1, se pasa a la fase de mejora de la misma mediante una búsqueda local. El procedimiento de búsqueda local propuesto está basado en movimientos de inserción. Dado un etiquetado f de los vértices de un grafo, se define un movimiento de inserción como $\texttt{INS}(f, j, v)$, donde v es el vértice que se mueve de su posición actual $f(v)$ y se inserta en la posición j. Esta operación da como resultado la ordenación f' de la siguiente manera:

- Si $f(v) = i > j$, el vértice v se inserta justo antes del vértice v_j en la posición j. En términos matemáticos, de la ordenación $f = (\ldots, v_{j-1}, v_j, v_{j+1}, \ldots, v_{i-1}, v, v_{i+1})$ se obtiene la ordenación $f' = (\ldots, v_{j-1}, v, v_j, v_{j+1}, \ldots, v_{i-1}, v_{i+1})$.

- Si $f(v) = i < j$, el vértice v se inserta justo después del vértice v_j en la posición j. En términos matemáticos, de la ordenación $f = (\ldots, v_{i-1}, v, v_{i+1}, \ldots, v_{j-1}, v_j, v_{j+1})$ se obtiene la ordenación $f' = (\ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_j, v, v_{j+1})$.

Se define el conjunto de vértices críticos (CV, del inglés *Critical Vertices*) como aquellos vértices cuyo valor del *cutwidth* es igual o próximo al *cutwidth* del grafo. Estos vértices determinan el valor de la función objetivo en la presente iteración o lo harán, con una alta probabilidad, en próximas iteraciones. En cada iteración, el procedimiento de búsqueda local selecciona un vértice $v \in CV$ y prueba distintos movimientos tipo $\texttt{INS}(f, j, v)$. El proceso de inserción de dicho vértice en distintas posiciones se detiene, cuando se han probado todos los posibles movimientos establecidos para el vértice, o cuando se produce un primer movimiento de mejora. En este caso, se mantiene el movimiento y el vértice es eliminado de CV. Las distintas posiciones j en el movimiento de inserción se determinan (para cada v) como un pequeño porcentaje del número de vértices del grafo. En concreto, se consideran las posiciones próximas a la mediana de las posiciones de los vértices adyacentes a v, de acuerdo con la ordenación f.

La lista CV se recalcula cuando se han explorado los posibles movimientos para todos los elementos de la misma, y alguno de ellos ha realizado un movimiento de mejora. El procedimiento de búsqueda local concluye cuando ningún movimiento de un vértice en CV se convierte en un movimiento de mejora.

Cabe destacar, que el concepto de movimiento de mejora no se limita únicamente a una mejora en el valor de la función objetivo, ya que esto restringiría mucho la búsqueda. Se considera un movimiento de mejora como aquél que reduce, o bien el $CW_f(G)$, o bien el número de vértices en CV.

Capítulo 4

Algoritmos heurísticos para la resolución del CMP

La resolución heurística de un problema de optimización persigue obtener soluciones de alta calidad en un tiempo de cómputo reducido. Este tipo de técnicas, junto con las metaheurísticas, se engloban dentro de las denominadas técnicas aproximadas. En este capítulo se propone la combinación de las metaheurísticas GRASP y Búsqueda Dispersa para la resolución del «problema de la minimización de la anchura de corte en ordenaciones lineales». Para ello, se proponen diferentes heurísticas constructivas y de búsqueda local, que son embebidas en las metaheurísticas anteriormente mencionadas. Se describen también, de manera detallada, los distintos métodos propuestos para la combinación de soluciones y para el cálculo de la estimación de la distancia entre las mismas.

4.1. Introducción a las técnicas de resolución aproximada

Las técnicas de resolución aproximada representan una alternativa a las técnicas de resolución exacta, a la hora de obtener soluciones a problemas de optimización. La alternativa ofrecida consiste en encontrar soluciones de alta calidad (que pueden llegar a ser, en ocasiones, incluso soluciones óptimas) en un tiempo de cómputo acotado. Los algoritmos que devuelven soluciones próximas al óptimo son llamados algoritmos de aproximación.

El principal inconveniente de las técnicas exactas, descritas en el Capítulo 3, es que pese a ser capaces de proporcionar soluciones óptimas, en general, requieren un tiempo de cómputo elevado cuando el tamaño del problema es grande. Numerosas aplicaciones prácticas enunciadas como problemas de optimización, requieren una respuesta en un tiempo mucho menor del que necesitan las técnicas exactas para encontrar la solución óptima.

Una posible clasificación de los distintos algoritmos de aproximación fue propuesta en [172], donde éstos se dividen en dos grandes grupos: algoritmos aproximados y algoritmos heurísticos. Los algoritmos aproximados permiten obtener soluciones cuya calidad es mensurable, ya que

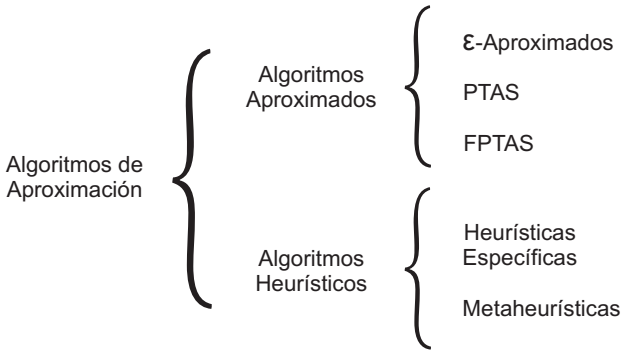


Figura 4.1: Clasificación de distintos algoritmos de aproximación (Adaptado de [172]).

proporcionan una cota respecto al valor de la solución óptima, es decir, ofrecen una garantía teórica de la cercanía al óptimo [87]. Estos algoritmos pueden clasificarse en «ε-Aproximados», «PTAS» (del inglés *Polynomial-Time Approximation Scheme*) y «FPTAS» (del inglés *Fully Polynomial-Time Approximation Scheme*). Por otro lado, los algoritmos heurísticos permiten obtener soluciones de alta calidad en tiempos de cómputo razonables, pero no permiten certificar si dichas soluciones son óptimas, ni tampoco la calidad que éstas atesoran. Los algoritmos heurísticos podrían, a su vez, subdividirse en «heurísticas específicas» y «metaheurísticas» [172]. En la Figura 4.1 se muestra, de manera esquemática, la clasificación de los distintos algoritmos de aproximación anteriormente mencionados.

4.1.1. Algoritmos aproximados

El estudio de algoritmos aproximados para la resolución de problemas de optimización ayuda a conocer la dificultad de los mismos y puede ser útil en el diseño de heurísticas eficientes. No obstante, los algoritmos aproximados presentan, en general, la desventaja de ser específicos para cada problema, lo que limita su aplicabilidad. Además, en la práctica, las aproximaciones alcanzables suelen estar lejos del óptimo global, lo que puede convertirlos en poco útiles para la resolución de ciertos problemas reales. Diseñar algoritmos aproximados para un problema de optimización tiene como objetivo encontrar cotas lo más ajustadas posible, en el peor caso.

Dado un problema de optimización cuyas soluciones puedan ser evaluadas como un coste positivo (con independencia de si el problema es de maximización o de minimización) se dice que un algoritmo para dicho problema tiene un ratio de aproximación $\rho(n)$ si, para cualquier entrada de tamaño n al algoritmo, el coste C de la solución producida por el algoritmo está relacionado mediante un factor $\rho(n)$ con el coste de la solución óptima C^* [42]:

$$\text{máx}\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

Un algoritmo que ofrece soluciones aproximadas con un ratio $\rho(n)$, es llamado algoritmo $\rho(n)$-

aproximado. Para un problema de maximización, se tiene que $0 < C \leq C^*$, siendo el ratio $\frac{C^*}{C}$ el factor por el cual el coste de una solución óptima es mayor que el de una solución aproximada. De manera similar, para un problema de minimización, se tiene que $0 < C^* \leq C$, siendo el ratio $\frac{C}{C^*}$ el factor por el cual el coste de una solución aproximada es mayor que el de una solución óptima. El ratio de un algoritmo aproximado nunca será menor que 1, siendo un algoritmo 1-aproximado aquél que produce una solución óptima. Por lo tanto, las soluciones encontradas por un algoritmo aproximado con un ratio mayor que 1, son peores que la solución óptima.

Los problemas de optimización pueden ser aproximados en tiempo polinómico de diferente manera. Para muchos problemas existen algoritmos aproximados con un ratio constante pequeño. Para otros problemas se dispone de ratios que crecen en función del tamaño de la entrada n. Por otro lado, existen también algoritmos aproximados que, para algunos problemas, pueden alcanzar ratios de aproximación más y más pequeños a medida que emplean más tiempo. Es decir, existe un compromiso entre el tiempo de ejecución y la calidad de la aproximación.

A continuación se repasan distintas clases de algoritmos aproximados:

- **ε-aproximados**: un algoritmo ε-aproximado es aquél capaz de generar una solución aproximada a, no menor que un factor ε veces la solución óptima s [179]. Para un problema de minimización, un algoritmo tiene un factor de aproximación ε si su complejidad temporal es polinómica y para cada instancia de entrada al algoritmo produce una solución tal que:

$$a \leq \varepsilon \cdot s \quad \text{si } \varepsilon > 1$$
$$\varepsilon \cdot s \leq a \quad \text{si } \varepsilon < 1$$

  donde s es una solución óptima y el factor ε ($\varepsilon > 0$) define el error acotado. Este factor puede ser una constante o bien una función del tamaño de la instancia de entrada [172]. Un algoritmo ε-aproximado genera un error acotado ε si se cumple la siguiente propiedad:

$$(s - \varepsilon) \leq a \leq (s + \varepsilon)$$

- **PTAS**: un esquema de aproximación para un problema de optimización, es un algoritmo aproximado que toma como entrada tanto una instancia del problema a resolver, como un valor $\varepsilon > 0$ tal que, para cualquier valor prefijado ε, el esquema es $(1+\varepsilon)$-aproximado. Se dice que un esquema de aproximación es un esquema PTAS si para cualquier $\varepsilon > 0$ prefijado, el esquema se puede ejecutar en tiempo polinómico para un tamaño n de la instancia de entrada [42].

- **FPTAS**: el tiempo de ejecución de un esquema aproximado se puede incrementar muy rápidamente a medida que ε decrece. Idealmente, si ε decrece en un factor constante, el

tiempo de ejecución no debería aumentar más de un factor también constante. Se dice que un esquema aproximado es un esquema FPTAS si tiene un algoritmo $(1+\varepsilon)$-aproximado tanto en términos del tamaño de la instancia de entrada n como de $1/\varepsilon$, para cualquier valor prefijado $\varepsilon > 0$ [42, 172].

4.1.2. Algoritmos heurísticos

Como se ha indicado anteriormente, los algoritmos heurísticos son métodos aproximados que permiten encontrar soluciones de alta calidad a problemas de optimización en tiempos de cómputo reducidos. Presentan la desventaja de no poder certificar si la solución encontrada es óptima, ni tampoco cómo de lejos se encuentra de ella. Es decir, sacrifican la garantía de la obtención de soluciones óptimas a cambio de una reducción ostensible en el tiempo de cómputo [19]. Los algoritmos heurísticos podrían subdividirse en heurísticas específicas y metaheurísticas:

- **Heurísticas específicas**: representan procedimientos diseñados y adaptados para resolver problemas concretos o instancias específicas. Suelen ser ideas sencillas y se basan en la construcción y mejora de soluciones. Su principal inconveniente es que no son capaces de escapar de óptimos locales. Dos tipos de heurísticas específicas ampliamente conocidas son las heurísticas constructivas y de las heurísticas de búsqueda local:

  - **Constructivas**: las heurísticas constructivas suelen ser los métodos aproximados más rápidos. Se encargan de la creación de soluciones partiendo, generalmente, de una solución parcial vacía y acabando con la formación de una solución completa factible.
  - **Búsqueda local**: las heurísticas de búsqueda local son métodos que parten de una solución inicial y tratan, en cada iteración, de sustituirla por otra solución mejor. En general requieren mucho más tiempo de cómputo que las heurísticas constructivas, si bien suelen ser capaces de obtener soluciones de mejor calidad que éstas.

  En la Sección 1.4.2 se repasan algunas de las estrategias constructivas y de mejora más conocidas.

- **Metaheurísticas**: las metaheurísticas son algoritmos de propósito general que pueden ser aplicados para resolver cualquier problema de optimización combinatoria. Se podrían ver como procedimientos generales de nivel superior capaces de establecer una estrategia que sirve de guía a otras heurísticas subyacentes en la resolución de problemas de optimización.

  En la actualidad, existen un gran número de técnicas metaheurísticas para las que encontrar una taxonomía capaz de clasificarlas de una manera eficaz y sin que existan solapamientos es una tarea compleja. No obstante, sí que es posible agrupar las metaheurísticas según determinados criterios que hacen referencia a alguna característica

particular de las mismas, como por ejemplo: si parten de una o múltiples soluciones, si emplean búsqueda local, si tienen inspiración biológica, si emplean más de una estructura de vecindad, si utilizan memoria, etc. Se pueden consultar diversas clasificaciones en [16, 18, 51, 65, 122].

Una posible clasificación basada en el número de soluciones de partida que son manejadas podría agrupar las metaheurísticas más importantes en poblacionales, trayectoriales y multiarranque, tal y como se muestra en la Figura 4.2. A continuación, se describen cada una de ellas:

- **Trayectoriales**: el término de «metaheurística trayectorial» nace del proceso de búsqueda llevado a cabo por estas metaheurísticas. En concreto, se podría decir que describen una trayectoria en el espacio de soluciones. Muchas de estas metaheurísticas son extensiones de procedimientos de búsqueda local, que por sí mismos resultan infructuosos. Algunas de las metaheurísticas trayectoriales más conocidas son: Búsqueda Tabú (TS, del inglés *Tabu Search*) [67, 68], Búsqueda de Vecindad Variable (VNS, del inglés *Variable Neighbourhood Search*) [126], Recocido Simulado (SA, del inglés *Simulated Annealing*) [96], etc.
- **Poblacionales**: las «metaheurísticas poblacionales» deben su nombre a que, en cada iteración del proceso de búsqueda, emplean un conjunto de soluciones (población) en lugar de únicamente una solución. El resultado final depende, en gran medida, de cómo se manipule dicha población. Algunas de las metaheurísticas poblacionales más conocidas son: Algoritmos Genéticos (GA, del inglés *Genetic Algorithm*) [88], Algoritmos Meméticos (MA, del inglés *Memetic Algorithm*) [131], Búsqueda Dispersa (SS, del inglés *Scatter Search*) [67, 71], Reencadenamiento de Trayectorias (PR, del inglés *Path Relinking*) [70, 73], etc.
- **Mutiarranque**: pese a que algunas taxonomías incluirían este tipo de metaheurísticas dentro de las trayectoriales, también pueden considerarse de manera independiente. Este tipo de metaheurísticas emplean un esquema basado en procedimientos de construcción y mejora de soluciones, tal y como lo hacen las metaheurísticas trayectoriales. Sin embargo, en lugar de partir de una única solución, realizan múltiples iteraciones que les permiten partir de diferentes soluciones. Algunas de las metaheurísticas multiarranque más conocidas son: Búsqueda Voraz Aleatorizada Adaptativa (GRASP, del inglés *Greedy Randomized Adaptive Search Procedure*) [55, 56], Concentración Heurística (HC, del inglés *Heuristic Concentration*) [160, 161], Multiarranque Adaptativo (AMS, del inglés *Adaptive Multi-Start*) [23], etc.

Las diferentes técnicas trayectoriales, poblacionales y multiarranque anteriormente mencionadas, son sólo algunas de las metaheurísticas existentes. Puede verse una

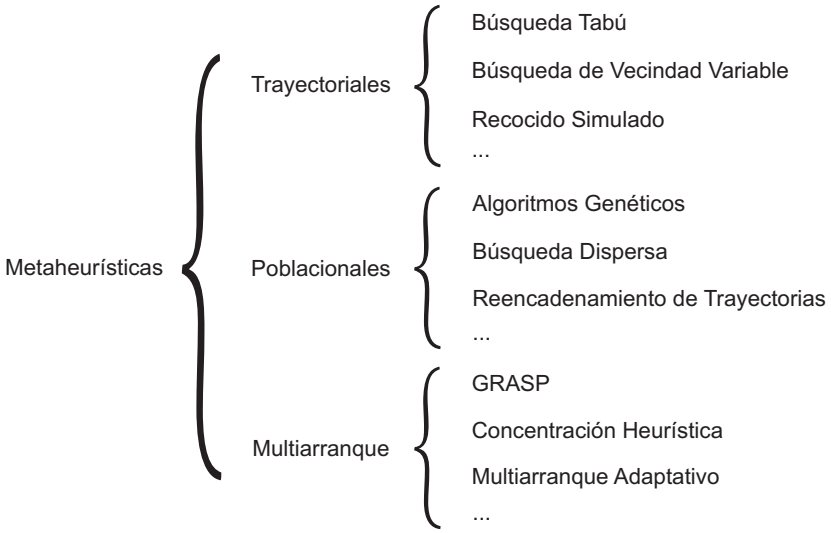


Figura 4.2: Clasificación de distintas metaheurísticas.

descripción más detallada de éstas y otras metaheurísticas en [18, 66, 72]. Para una descripción de las metaheurísticas Búsqueda Dispersa y GRASP, empleadas en esta Tesis Doctoral para la resolución del CMP, puede consultarse también la Sección 1.4.2.

4.2. Métodos previos

En el Capítulo 2 de esta Tesis Doctoral se lleva a cabo un repaso de los trabajos previos relacionados con el CMP. En cuanto a la resolución aproximada del problema se refiere, se han encontrado tanto algoritmos aproximados como algoritmos heurísticos que abordan el CMP. Se presenta un breve repaso de estos algoritmos en las secciones 4.2.1 y 4.2.2 respectivamente.

4.2.1. Algoritmos aproximados para la resolución del CMP

Se han encontrado únicamente dos trabajos en los que se proponen algoritmos aproximados para la resolución del CMP:

- En [7] se propone un algoritmo tipo PTAS para grafos densos con una complejidad temporal $n^{O\left(1/\varepsilon^2\right)}$, considerando un grafo con n vértices y m aristas como denso si $m = \Theta(n^2)$.

- En [103] se propone un algoritmo $O(\log^2 n)$-aproximado, siendo n el número de vértices del grafo de entrada. Este algoritmo nace como aplicación directa de otro algoritmo previo $O(\log\ n)$-aproximado destinado a encontrar particiones balanceadas en un grafo.

4.2.2. Algoritmos heurísticos para la resolución del CMP

En cuanto a la resolución heurística del CMP se refiere, se han encontrado diversos trabajos en los que se aborda el problema empleando este tipo de algoritmos [3, 2, 38, 40, 153]. A pesar de haber encontrado un total de cinco publicaciones, dichos trabajos se corresponden únicamente con dos propuestas diferentes, ya que algunas de las publicaciones encontradas son versiones preliminares de otras:

- En [38, 40] se aborda una generalización del CMP denominada *Board Permutation Problem*, donde se proponen diversas heurísticas constructivas y un conjunto de métodos de mejora. Los autores emplean la técnica metaheurística del Recocido Simulado para combinar su mejor estrategia constructiva y su mejor método de mejora.
- Los autores de [3, 2] abordan el CMP para resolver una aplicación práctica del problema (*Network Migration Scheduling*) que consistente en migrar una serie de dispositivos conectados entre sí, de una red a otra. Para ello, inicialmente plantean un algoritmo tipo GRASP combinado con un mecanismo de Reencadenamiento de Trayectorias. Posteriormente, extienden el trabajo anterior mediante la combinación de GRASP con Reencadenamiento de Trayectorias Evolutivo. Se puede consultar una versión completa del algoritmo propuesto en una patente publicada recientemente [153].

Los algoritmos heurísticos para la resolución del CMP presentes en el estado del arte, se consideran de especial relevancia para el desarrollo de esta Tesis Doctoral. Esto es debido a que siguen el mismo enfoque para la obtención de soluciones aproximadas al problema que el propuesto en esta Tesis Doctoral y que es desarrollado en este capítulo. Dichos algoritmos son utilizados como marco de comparación para los algoritmos propuestos. Por ello, se presenta una descripción detallada de ambas propuestas en las secciones 2.4.1 y 2.4.2.

4.3. Algoritmos constructivos

En esta sección se proponen cuatro heurísticas constructivas diferentes para la obtención de soluciones al CMP. Dichas heurísticas están basadas en la metodología GRASP [55, 56, 154] y podrían ser consideradas como una evolución del algoritmo constructivo presentado en la Sección 3.4.1.

El objetivo inicial de los algoritmos constructivos propuestos (denominados C1, C2, C3 y C4) es proporcionar soluciones iniciales factibles al CMP, de modo que éstas tengan la mejor calidad posible. No obstante, tal y como se verá más adelante, también es relevante la diversidad de las soluciones producidas, ya que el objetivo final de los algoritmos constructivos es que puedan integrarse en un esquema de SS.

A la hora de diseñar un algoritmo constructivo es importante tener en cuenta el tipo de solución del problema abordado. En este caso, cabe destacar que, dada la estructura de una solución para el CMP, cualquier ordenación de los vértices del grafo de entrada representa una solución factible al problema. A continuación se repasan las heurísticas constructivas propuestas.

4.3.1. Constructivo 1 (C1)

Al igual que el método propuesto en la Sección 3.4.1, el algoritmo constructivo C1 comienza con la creación de una lista de vértices no etiquetados U, que inicialmente contiene todos los vértices del grafo ($U = V$, siendo V el conjunto de vértices del grafo) y un conjunto de vértices etiquetados L que inicialmente está vacío ($L = \varnothing$).

En el primer paso del algoritmo, todos los vértices de U son candidatos, por lo que el procedimiento constructivo escoge, siguiendo un criterio voraz, un vértice $v \in U$ y le asigna la etiqueta 1. El criterio voraz empleado para seleccionar el vértice que ocupará la primera posición de la ordenación está basado en el grado de cada vértice candidato. En concreto, se selecciona el vértice con un menor número de adyacentes. Si más de un vértice candidato tuviera el menor número de adyacentes, se escogería uno de ellos de manera aleatoria.

En la Figura 4.3.a se puede ver un ejemplo de grafo $G(V, E)$ con 6 vértices y 10 aristas. En la Figura 4.3.b se muestran las seis posibles soluciones parciales, en las que sólo está asignada la primera posición de la ordenación, para el grafo G. Tal y como se estudió en el Capítulo 3, se puede determinar el valor del *cutwidth* que tendrán los vértices asignados a una solución parcial, cuando ésta se complete. Como se puede observar en las distintas soluciones parciales de la Figura 4.3.b, el *cutwidth* del vértice situado en la primera posición es diferente en función del vértice escogido para ocuparla. De manera intuitiva parece favorable escoger aquellas soluciones parciales que, de partida, tienen un menor valor del *cutwidth*. Este hecho se produce en aquellas ordenaciones que comienzan por los vértices de menor grado, en concreto los vértices E y F. Por el contrario, el vértice de mayor grado, A, es el que genera un mayor valor del *cutwidth*, al situarlo en la primera posición de la ordenación.

En las siguientes iteraciones de C1 se asignan las etiquetas disponibles, una a una, siguiendo un orden lexicográfico. En cada iteración del algoritmo sólo será necesario, por lo tanto, decidir qué vértice será el siguiente en ser etiquetado. Para ello, el algoritmo construye una lista de candidatos (CL, del inglés *Candidate List*) que incluye todos los vértices de U con al menos un vértice adyacente etiquetado. Para cada vértice $u \in CL$ se evalúa su *cutwidth*, $CW_f(u)$, al ocupar la posición k, considerando que k es la etiqueta más baja disponible, tal que $f(u) = k$. Para evaluar $CW_f(u)$ es necesario conocer el número de vértices adyacentes a u previamente etiquetados, $N_L(u)$, y el número de vértices adyacentes a u que todavía no han sido etiquetados, $N_U(u)$, además del *cutwidth* de la última posición etiquetada, $CW_f(v)$. En términos matemáticos se puede calcular como:

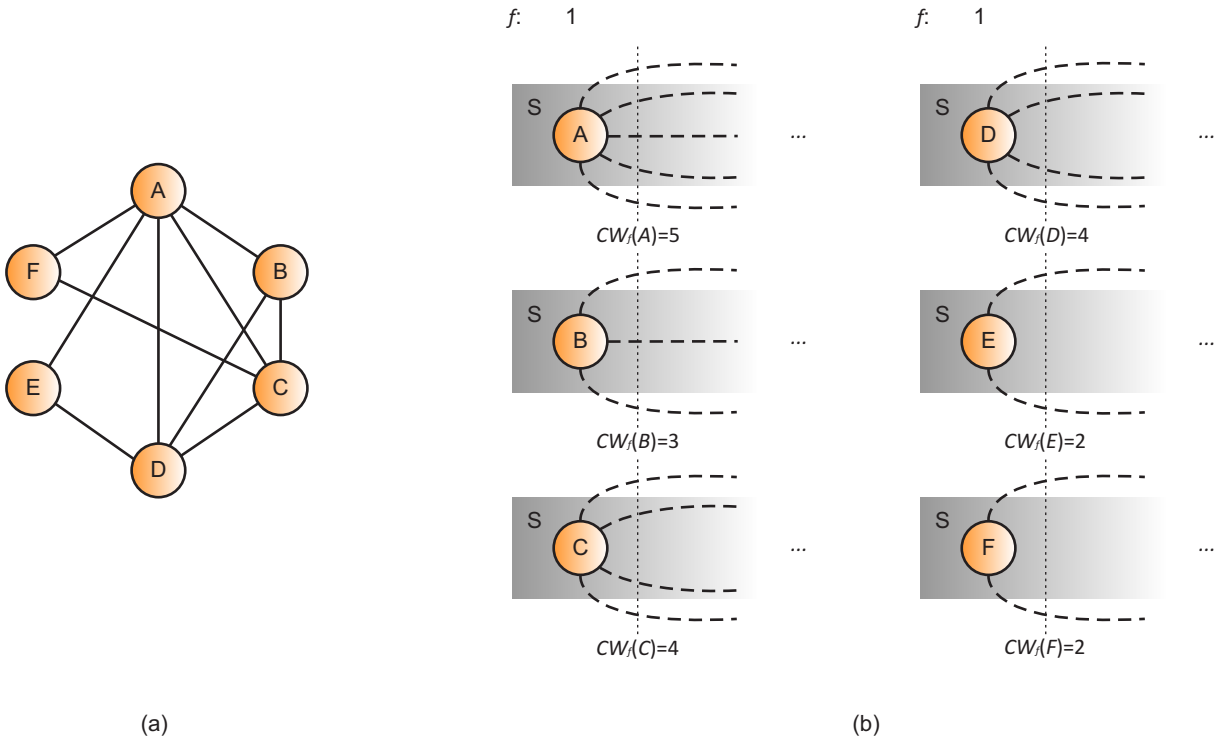
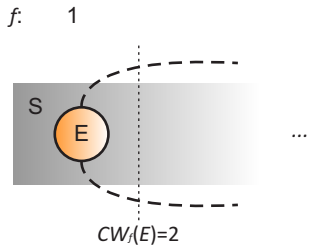


Figura 4.3: (a) Representación de un grafo $G = (V, E)$. (b) Distintas soluciones parciales posibles, en la asignación de la primera posición a los vértices de G.

$$CW_f(u) = CW_f(v) - \mid N_L(u) \mid + \mid N_U(u) \mid .$$

Supóngase que, para el grafo de la Figura 4.3.a, el vértice escogido de manera aleatoria (entre los vértices E y F) para ocupar la primera posición de la ordenación f es el vértice E (ver Figura 4.4.a). En ese caso, los vértices candidatos a ocupar la segunda posición de f serían los vértices adyacentes a E, es decir, A y D. Por lo tanto, $CL = \{A, D\}$. En la Figura 4.4.b se muestra la evaluación de cada uno de los vértices candidato, en el caso de que ocuparan la siguiente posición disponible $k = 2$. Se puede observar que $CW_f(E) = 2$ y tanto A como D tienen únicamente un adyacente etiquetado, por lo que su evaluación dependerá del número de adyacentes no etiquetados que tenga cada uno. En concreto, situar el vértice A en la posición $k = 2$ resultaría en $CW_f(A) = 5$, mientras que situar el vértice D, resultaría en $CW_f(D) = 4$.

En esta situación, un algoritmo voraz seleccionaría el vértice $u \in CL$ con un menor valor $CW_f(u)$, es decir, seleccionaría el vértice D para ser el siguiente en ser etiquetado, ya que $CW_f(D) < CW_f(A)$. En cambio, el algoritmo C1 (tipo GRASP) en lugar de escoger el vértice más prometedor de manera voraz, escoge aleatoriamente un vértice entre el subconjunto de vértices más prometedores de CL. Dicho subconjunto es denominado Lista de Candidatos Restringida (RCL, del inglés *Restricted Candidate List*) y se forma con aquellos vértices u, cuyo $CW_f(u)$ sea menor o igual que un determinado umbral th. Este umbral permite seleccionar



CL	$CW_f(v)$ - $\lvert N_L(u)\rvert$ - $\lvert N_U(u)\rvert$
A	2 - |1| + |4| = 5
D	2 - |1| + |3| = 4

(a) (b)

Figura 4.4: (a) Solución parcial con el vértice E asignado a la posición 1 ($f(E) = 1$). (b) Evaluación de los vértices de la lista de candidatos a ocupar la posición 2 en la ordenación f.

un porcentaje de los vértices más prometedores de CL. Para ello, los algoritmos tipo GRASP suelen emplear un parámetro de búsqueda $\alpha \in [0, 1]$. En términos matemáticos, th se podría calcular de la siguiente manera:

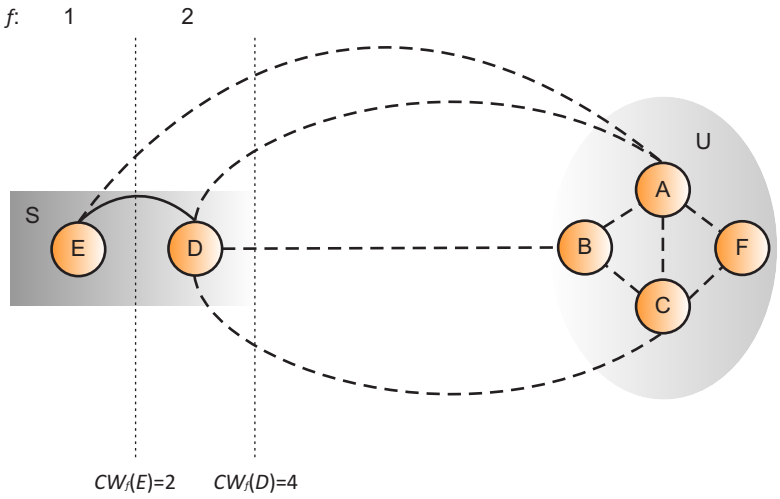
$$th = CW_{min} + \alpha(CW_{max} - CW_{min})$$

donde CW_{min} y CW_{max} representan, respectivamente, al menor y al mayor $CW_f(u)$, $\forall\, u \in CL$. El valor del parámetro α se selecciona de manera aleatoria en cada ejecución de C1. Cabe destacar que si $\alpha = 0$ la selección del siguiente vértice a etiquetar es completamente voraz entre los vértices de CL, mientras que si $\alpha = 1$ dicha selección es completamente aleatoria.

Supóngase que, para el ejemplo de la Figura 4.4, donde $CL = \{A, D\}$, se tiene un valor $\alpha = 0{,}7$. En este caso, $th = 4 + 0{,}7(5 - 4) = 4{,}7$, luego la RCL estaría formada por todos aquellos vértices de la CL cuyo $CW_f(v)$ fuera menor que $4{,}7$, es decir, $RCL = \{D\}$. Esto implica que $f(D) = 2$, obteniendo una solución parcial como la mostrada en la Figura 4.5.a. En dicha figura, $U = \{A, B, C, F\}$ (vértices aún sin etiquetar) y $CL = \{A, B, C\}$ (vértices candidatos, por ser adyacentes a algún vértice etiquetado).

En la Figura 4.5.b se muestra la evaluación de cada vértice $u \in CL$, como candidatos a formar parte de la RCL en la siguiente iteración del algoritmo. En este caso, entran en la RCL aquellos u cuyo $CW_f(u)$ sea menor o igual que $th = 5 + 0{,}7(6 - 5) = 5{,}7$. En concreto, $RCL = \{A, B\}$, luego el siguiente vértice a etiquetar se escogería aleatoriamente entre los vértices A y B. El proceso se repite hasta que todos los vértices de U hayan sido etiquetados.

El pseudocódigo del método constructivo C1 se muestra en Algoritmo 4.1. Como se ha indicado anteriormente, en dicho pseudocódigo, L (inicialmente vacío) representa al conjunto de todos los vértices etiquetados del grafo y, el U (inicialmente $U = V$, siendo V el conjunto de vértices del grafo) representa al conjunto de vértices que aún no han sido etiquetados. El algoritmo comienza escogiendo un vértice de mínima adyacencia (paso 4) y le asigna la etiqueta 1 (paso 5). A continuación, se elimina el vértice escogido del conjunto U y se añade al conjunto



CL	$CW_f(v) - \lvert N_L(u)\rvert - \lvert N_U(u)\rvert$
A	4 - |2| + |3| = 5
B	4 - |1| + |2| = 5
C	4 - |1| + |3| = 6

Figura 4.5: (a) Solución parcial con el vértice E asignado a la posición 1 ($f(E) = 1$) y el vértice D asignado a la posición 2 ($f(D) = 2$). (b) Evaluación de los vértices de la lista de candidatos a ocupar la posición 3 en la ordenación f.

L (paso 6). En cada iteración, comprendida entre los pasos 7 y 17, se etiqueta un vértice del grafo hasta que todos ellos tienen una etiqueta asignada. Para escoger el siguiente vértice a etiquetar, en cada iteración, se forma una CL con todos los vértices no etiquetados que tengan un vértice adyacente etiquetado (paso 9). Después, se calcula el *cutwidth* que tendría cada uno de los vértices candidatos, en el caso de que éstos fueran etiquetados con la siguiente etiqueta disponible (paso 10). Una vez calculado, se establece un umbral basado en el valor mínimo y máximo de dicho *cutwidth* (pasos 11 y 12) que determina qué vértices de la CL pasan a formar parte de la RCL (paso 13). Cuando la RCL está construida, se selecciona un vértice de ella de manera aleatoria (paso 14) y se etiqueta con la etiqueta más baja aún disponible (paso 15). Por último, en el paso 16, el vértice etiquetado se elimina del conjunto U y se añade al conjunto L. El proceso se repite hasta que todos los vértices del grafo tienen una etiqueta asignada.

4.3.2. Constructivo 2 (C2)

El segundo algoritmo constructivo propuesto para la obtención de soluciones al CMP, denominado C2, es una variante de la metodología GRASP basada en las ideas propuestas en [157]. Al igual que C1, el algoritmo comienza asignando la etiqueta 1 al vértice de menor grado del grafo o, si hubiera más de uno, a uno de los de menor grado escogido aleatoriamente. En cada una de las siguientes iteraciones del algoritmo se van escogiendo los vértices no etiquetados, uno a uno, y se les asigna la etiqueta más baja aún disponible, es decir, en orden lexicográfico.

Para determinar el siguiente vértice a etiquetar en cada iteración, de manera también similar a C1, se construye una lista de candidatos (CL) con aquellos vértices que sean adyacentes a algún

Algoritmo 4.1 Algoritmo constructivo C1

1: **procedimiento** ConstructivoC1
2: Sean L y U los conjuntos de vértices etiquetados y no etiquetados de un grafo respectivamente;
3: Inicialmente $L = \varnothing$ y $U = V$;
4: $u = \text{mín}_{v \in V}\{\mid N(v) \mid\}$;
5: Asignar la etiqueta $k = 1$ a u $(f(u) = 1)$;
6: $L = \{u\}$, $U = U \setminus \{u\}$;
7: **mientras** $U \neq \varnothing$ **hacer**
8: $k = k + 1$;
9: Construir $CL = \{u \in U \,/\, (w, v) \in E, \forall\, w \in L\}$;
10: Calcular $CW_f(u)$, $\forall\, u \in CL$ considerando que u es etiquetado con la etiqueta k $(f(u) = k)$;
11: Calcular $CW_{min} = \text{mín}_{u \in CL}\, CW_f(u)$ y $CW_{max} = \text{máx}_{u \in CL}\, CW_f(u)$;
12: Calcular $th = CW_{min} + \alpha(CW_{max} - CW_{min})$;
13: Construir $RCL = \{u \in CL \,/\, CW_f(u) \leq th\}$;
14: Seleccionar un vértice u^* de manera aleatoria de RCL;
15: Etiquetar u^* con la etiqueta k $(f(u^*) = k)$;
16: $U = U \setminus \{u^*\}, L = L \cup \{u^*\}$;
17: **fin mientras**;
18: **fin** ConstructivoC1;

vértice previamente etiquetado. La diferencia con el algoritmo C1 reside en la formación de la RCL a partir de la CL. En concreto, la RCL estará formada por un número Tam de vértices de la CL, escogidos de manera aleatoria. El tamaño de la RCL se establece como un porcentaje del tamaño de la CL, empleando para ello un parámetro $\alpha \in [0, 1]$, siendo $Tam = \alpha \mid CL \mid$.

Una vez determinado el tamaño de la RCL y los vértices $u \in CL$ que forman la RCL, se evalúa $CW_f(u)$, $\forall\, u \in RCL$, suponiendo que se etiquetara u con la menor etiqueta disponible k. Tras evaluar todos los vértices de la RCL, se escoge el vértice u^* con un *cutwidth* menor o, en caso de empate, se escoge uno de manera aleatoria entre los que tengan un menor *cutwidth*, y se le asigna la etiqueta k, de modo que $f(u) = k$. El proceso finaliza cuando todos los vértices del grafo tienen una etiqueta asignada. De esta manera, C2 intercambia el orden en el que las etapas de selección voraz y aleatoria son llevadas a cabo en C1.

El pseudocódigo del método constructivo C2 se muestra en Algoritmo 4.2. La principal diferencia con el Algoritmo 4.1 se encuentra entre los pasos 10 y 13, donde se construye la RCL y se selecciona el siguiente vértice a etiquetar. Concretamente, en el paso 11, se toman aleatoriamente Tam vértices de la CL para formar la RCL; en el paso 12, dichos vértices son evaluados y, en el paso 13, se selecciona el siguiente vértice a etiquetar como aquél, de entre los vértices de la RCL, que produzca un *cutwidth* menor al ser etiquetado con la siguiente etiqueta más baja disponible. Para una descripción más detallada del resto de pasos, puede verse la explicación realizada la Sección 4.3.1 para el algoritmo constructivo C1.

Algoritmo 4.2 Algoritmo constructivo C2

1: **procedimiento** ConstructivoC2
2: Sean L y U los conjuntos de vértices etiquetados y no etiquetados de un grafo respectivamente;
3: Inicialmente $L = \varnothing$ y $U = V$;
4: $u = \text{mín}_{v \in V}\{\mid N(v) \mid\}$;
5: Asignar la etiqueta $k = 1$ a u $(f(u) = 1)$;
6: $L = \{u\}$, $U = U \setminus \{u\}$;
7: **mientras** $U \neq \varnothing$ **hacer**
8: $k = k + 1$;
9: Construir $CL = \{u \in U \,/\, (w, v) \in E, \forall\, w \in L\}$;
10: $Tam = \alpha \mid CL \mid$;
11: Construir RCL seleccionando aleatoriamente Tam vértices de CL;
12: Calcular $CW_f(u)$, $\forall\, u \in RCL$ considerando que u es etiquetado con la etiqueta k $(f(u) = k)$;
13: Seleccionar el vértice $u^* = \text{mín}_{u \in RCL}\{CW_f(u)\}$;
14: Etiquetar u^* con la etiqueta k $(f(u^*) = k)$;
15: $U = U \setminus \{u^*\}, L = L \cup \{u^*\}$;
16: **fin mientras**;
17: **fin** ConstructivoC2;

4.3.3. Constructivo 3 (C3)

El tercer algoritmo constructivo propuesto, denominado C3, es también un algoritmo basado en la metodología GRASP. Se diferencia del algoritmo C1 en la función que emplea para evaluar qué vértices candidatos pasan a formar parte de la RCL. Concretamente, reemplaza la función de evaluación $CW_f(u)$, basada en la función objetivo del vértice a etiquetar (u) en la solución parcial (f), por la función $|N_L(u)|$, que representa el número de vértices adyacentes, previamente etiquetados, de cada vértice candidato u.

Esta estrategia está basada en la idea de que los vértices adyacentes deben estar lo más cerca posible en la ordenación, minimizando así el número de vértices que se encuentran entre ellos. Dado un vértice u, que ocupa una posición $f(u)$, y un vértice adyacente v, situado en la posición $f(v)$, el objetivo es reducir al máximo el número de vértices w tales que $f(u) < f(w) < f(v)$. Es decir, el algoritmo C3 trata de reducir el número de vértices que se verían afectados por la arista (u, v), mediante el etiquetado de vértices adyacentes con etiquetas próximas.

Además, dado que en este caso tienen preferencia aquellos vértices con un mayor número de adyacentes etiquetados (y no con un menor valor de la función objetivo, como sucedía en C1) se debe introducir una modificación en el cálculo del umbral th, que quedaría especificado de la siguiente manera: $th = N_{Lmax} - \alpha(N_{Lmax} - N_{Lmin})$, donde N_{Lmax} y N_{Lmin} representan, respectivamente, el número máximo y mínimo de adyacentes etiquetados de un vértice $v \in CL$. El parámetro $\alpha \in [0, 1]$ establece, nuevamente, el porcentaje de vértices de CL que se añadirán a la RCL.

El pseudocódigo del método constructivo C3 se muestra en Algoritmo 4.3. La principal diferencia con los algoritmos Algoritmo 4.1 y Algoritmo 4.2 se encuentra entre los pasos 10 y 14. Concretamente, en el paso 10 se computa el número de adyacentes etiquetados que tiene cada vértice candidato. En los pasos 11 y 12 se establece un umbral para determinar qué vértices de la CL pueden pasar a formar parte de la RCL (formada en el paso 13). Por último, en el paso 14, se selecciona uno de los vértices de la RCL de manera aleatoria. Para una descripción más detallada del resto de pasos, puede consultarse la explicación presentada en la Sección 4.3.1 para el algoritmo constructivo C1.

Algoritmo 4.3 Algoritmo constructivo C3

1: **procedimiento** ConstructivoC3
2: Sean L y U los conjuntos de vértices etiquetados y no etiquetados de un grafo respectivamente;
3: Inicialmente $L = \varnothing$ y $U = V$;
4: $u = \text{mín}_{v \in V}\{\mid N(v) \mid\}$;
5: Asignar la etiqueta $k = 1$ a u $(f(u) = 1)$;
6: $L = \{u\}$, $U = U \setminus \{u\}$;
7: **mientras** $U \neq \varnothing$ **hacer**
8: $k = k + 1$;
9: Construir $CL = \{u \in U \,/\, (w, v) \in E, \forall\, w \in L\}$;
10: Computar $\mid N_L(u) \mid$, $\forall\, u \in CL$;
11: Establecer $N_{Lmin} = \text{mín}_{u \in CL} \mid N_L(u) \mid$ y $N_{Lmax} = \text{máx}_{u \in CL} \mid N_L(u) \mid$;
12: Calcular $th = N_{Lmax} - \alpha(N_{Lmax} - N_{Lmin})$;
13: Construir $RCL = \{u \in CL \,/\, \mid N_L(u) \mid \geq th\}$;
14: Seleccionar un vértice u^* de manera aleatoria de RCL;
15: Etiquetar u^* con la etiqueta k $(f(u^*) = k)$;
16: $U = U \setminus \{u^*\}, L = L \cup \{u^*\}$;
17: **fin mientras**;
18: **fin** ConstructivoC3;

4.3.4. Constructivo 4 (C4)

El último algoritmo constructivo propuesto para la obtención de soluciones al CMP, denominado C4, es una variante del algoritmo constructivo C3, en el cual el orden de las etapas de selección voraz y aleatoria de la metodología GRASP está intercambiado (tal y como se realizó entre C1 y C2).

El algoritmo C4 emplea, al igual que el algoritmo C3, una función basada en el número de adyacentes etiquetados, $|N_L(u)|$, durante el proceso de selección del siguiente vértice a etiquetar. La diferencia fundamental entre el algoritmo C3 y el algoritmo C4, reside en que C3 inicialmente evalúa los vértices pertenecientes a CL (empleando la función anteriormente descrita) y, a continuación, añade un porcentaje de los mejores a la RCL, mientras que C4, escoge de manera aleatoria un subconjunto de vértices candidatos de la CL y los añade a la RCL.

Una vez que la RCL está formada, el algoritmo C4 evalúa $\mid N_L(u) \mid, \forall u \in RCL$ y escoge aquel

u cuyo $\mid N_L(u) \mid$ sea máximo. Este algoritmo está basado nuevamente en los principios descritos en la Sección 4.3.3 en cuanto a la posición relativa de un vértice respecto a sus adyacentes.

El pseudocódigo del método constructivo C4 se muestra en Algoritmo 4.4. Este algoritmo tiene similitudes con el Algoritmo 4.3. La diferencia fundamental está en el criterio seguido para formar la RCL y en el criterio seguido para escoger el siguiente vértice a etiquetar. Concretamente, Tam representa el número de vértices pertenecientes a CL que serán añadidos a la RCL, es decir, el tamaño de la RCL, que viene determinado como un porcentaje ($\alpha \in [0, 1]$) del tamaño de la CL. Una vez formada la RCL (paso 11) se calcula el número de adyacentes etiquetados que tiene cada vértice en la RCL (paso 12) escogiendo aquél con un número de adyacentes mayor (paso 13) como el siguiente vértice en ser etiquetado.

Algoritmo 4.4 Algoritmo constructivo C4

1: **procedimiento** ConstructivoC4
2: Sean L y U los conjuntos de vértices etiquetados y no etiquetados de un grafo respectivamente;
3: Inicialmente $L = \varnothing$ y $U = V$;
4: $u = \text{mín}_{v \in V}\{\mid N(v) \mid\}$;
5: Asignar la etiqueta $k = 1$ a u ($f(u) = 1$);
6: $L = \{u\}$, $U = U \setminus \{u\}$;
7: **mientras** $U \neq \varnothing$ **hacer**
8: $k = k + 1$;
9: Construir $CL = \{u \in U \,/\, (w, v) \in E, \forall\, w \in L\}$;
10: $Tam = \alpha \mid CL \mid$;
11: Construir RCL seleccionando aleatoriamente Tam vértices de CL;
12: Calcular $\mid N_L(u) \mid$, $\forall\, u \in RCL$;
13: Seleccionar el vértice $u^* = \text{máx}_{u \in RCL}\{\mid N_L(u) \mid\}$;
14: Etiquetar u^* con la etiqueta k ($f(u^*) = k$);
15: $U = U \setminus \{u^*\}, L = L \cup \{u^*\}$;
16: **fin mientras**;
17: **fin** ConstructivoC4;

4.4. Método de mejora

Un método de mejora es un procedimiento que, partiendo de una solución a un problema de optimización, trata de encontrar soluciones de mejor calidad en la vecindad de la misma. Una «búsqueda local» es un tipo de procedimiento heurístico de mejora en el que, en cada iteración, se sustituye la solución de partida por una solución de su vecindad, de mejor calidad, siendo esta última la nueva solución de partida para la siguiente iteración del algoritmo. Cuando la búsqueda local no puede encontrar una solución de mejor calidad en la vecindad definida, ésta habría encontrado un óptimo local y finalizaría su ejecución. El principal inconveniente de los procedimientos de búsqueda local es, precisamente, que no son capaces de escapar de óptimos locales.

Es común emplear búsquedas locales en combinación con algoritmos constructivos, de manera que la búsqueda local actúa como una etapa posterior al procedimiento constructivo. Inicialmente, el constructivo genera soluciones que servirán como punto de partida para la búsqueda local. Las soluciones encontradas por las búsquedas locales suelen ser de mejor calidad que aquéllas encontradas por las heurísticas constructivas, si bien también suelen emplear un mayor tiempo de cómputo para alcanzarlas.

En esta Tesis Doctoral se propone un procedimiento heurístico de búsqueda local para el CMP basado en movimientos de inserción. Este método de mejora es una evolución del método previo propuesto en la Sección 3.4.2.

4.4.1. Movimientos en una solución

El concepto de movimiento, referido a una solución s de un problema de optimización, se podría definir, de manera intuitiva, como una alteración de s mediante la que se obtiene una nueva solución s' situada en la vecindad de s.

De manera más formal, se define el término «estructura de vecindad» como una función $N : S \rightarrow 2^S$ que asigna a cada $s \in S$ un conjunto de vecinos $N(s) \subseteq S$, representando $N(s)$ la vecindad de la solución s [19]. A menudo, las estructuras de vecindad se definen de manera implícita, al especificar los cambios que deben ser aplicados a una solución s para alcanzar todos sus vecinos. La aplicación del operador que produce un vecino $s' \in N(s)$ a partir de una solución s, se denomina movimiento.

Cuando la estructura de la solución a un problema de optimización es una ordenación, como es el caso del CMP, se consideran típicamente dos tipos de movimientos: intercambios e inserciones. Los movimientos de intercambio en una ordenación f quedan definidos por dos vértices, y consisten en el intercambio de las etiquetas asignadas a cada vértice que interviene en el movimiento. En este caso, únicamente cambian su posición los dos vértices que definen el intercambio. En cambio, los movimientos de inserción se definen mediante un vértice y una posición de la ordenación en la que insertar el vértice. En este caso, cambian su posición todos los vértices situados entre la posición del vértice que es insertado y la posición destino de dicho vértice.

Es importante destacar que tras un movimiento, ya sea de inserción o de intercambio, es posible que, además de los vértices que lo definen, existan otros vértices de la ordenación que cambien su *cutwidth*. En concreto, para movimientos de intercambio, los vértices susceptibles a cambiar su *cutwidth* son aquéllos situados entre las posiciones de los vértices intercambiados. En el caso de los movimientos de inserción, los vértices cuyo *cutwidth* puede cambiar son aquéllos situados entre la posición original del vértice que se inserta y la posición destino del movimiento.

En esta Tesis Doctoral se propone una búsqueda local basada en movimientos de inserción

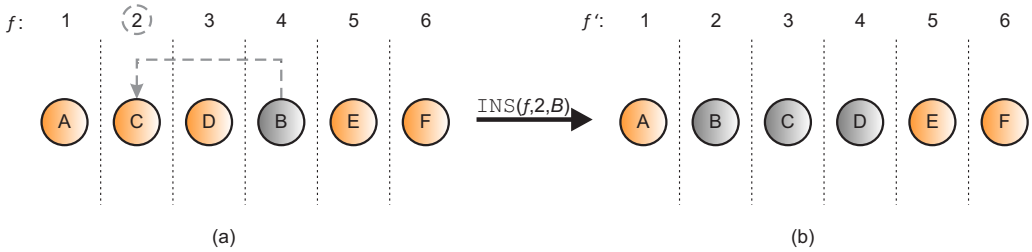


Figura 4.6: (a) Ordenación f de los vértices de un grafo. (b) Ordenación f' obtenida tras el movimiento $\texttt{INS}(f, 2, B)$.

para el CMP. A continuación, se detalla cómo se realiza un movimiento de inserción, así como la revaluación posterior de una solución.

Movimientos de inserción

De manera específica, dado un etiquetado f de los vértices de un grafo G, se define el movimiento de inserción $\texttt{INS}(f, j, v_i)$ como la acción de eliminar el vértice v_i de su posición actual $i = f(v_i)$ e insertarlo en la posición j, obteniendo una nueva ordenación f'. A tal efecto, se distinguen dos casos:

- Si $f(v) = i > j$, entonces el vértice v se inserta justo antes del vértice v_j en la posición j. En términos más formales, de la ordenación $f = (\ldots, v_{j-1}, v_j, v_{j+1}, \ldots, v_{i-1}, v, v_{i+1})$ se obtendría la ordenación $f' = (\ldots, v_{j-1}, v, v_j, v_{j+1}, \ldots, v_{i-1}, v_{i+1})$. En la Figura 4.6.a se muestra la representación de una ordenación f de los vértices de un grafo G y, en la Figura 4.6.b, se muestra la ordenación f' obtenida como resultado del movimiento de inserción $\texttt{INS}(f, 2, B)$. En este caso, la posición del vértice B en la ordenación f ($f(B) = 4$) es mayor que la posición 2, destino del movimiento. Como se muestra en la Figura 4.6.b, los vértices comprendidos entre las posiciones 2 y 4 se han visto afectados por el movimiento. El vértice B ha sido insertado en la posición 2 y los vértices C y D se han desplazado una posición a la derecha en la ordenación.

- Si $f(v) = i < j$, entonces el vértice v se inserta justo después del vértice v_j en la posición j. En términos más formales, de la ordenación $f = (\ldots, v_{i-1}, v, v_{i+1}, \ldots, v_{j-1}, v_j, v_{j+1})$ se obtendría la ordenación $f' = (\ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_j, v, v_{j+1})$. En la Figura 4.7.a se muestra la representación de una ordenación f de los vértices de un grafo G y, en la Figura 4.7.b, se muestra la ordenación f' como resultado del movimiento de inserción $\texttt{INS}(f, 5, C)$. En este caso, la posición del vértice C en la ordenación f ($f(C) = 3$) es menor que la posición 5, destino del movimiento. Como se muestra en la Figura 4.7.b, los vértices comprendidos entre las posiciones 3 y 5 se han visto afectados por el movimiento. El vértice C ha sido insertado en la posición 5 y los vértices D y E se han desplazado una posición a la izquierda en la ordenación.

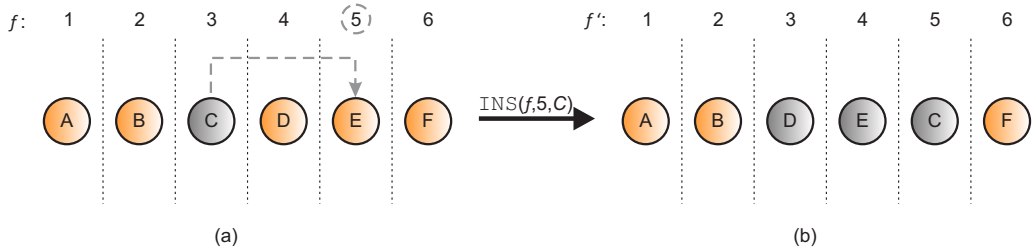


Figura 4.7: (a) Ordenación inicial f de los vértices de un grafo. (b) Ordenación f' tras el movimiento $\texttt{INS}(f, 5, C)$.

En las figuras 4.6 y 4.7 se representan sendas ordenaciones de los vértices de un grafo antes y después de un movimiento de inserción. No obstante, en dichas figuras no se valora el impacto en la función objetivo de tales movimientos. Al realizar un movimiento, es posible que el *cutwidth* de cada uno de los vértices afectados por el mismo se vea modificado, ya que las aristas de dichos vértices pueden pasar a afectar a otros vértices. Además, también es posible que el valor de la función objetivo de la nueva solución obtenida sea distinto al valor de la función objetivo de la solución original. En la Figura 4.8.a se muestra la ordenación f de los vértices del grafo $G(V, E)$ de la Figura 4.3.a, con sus aristas correspondientes. En la misma figura, se muestra también la evaluación de $CW_f(v)\ \forall\ v \in V$, para cada vértice que ocupa una posición evaluable de la ordenación f. En la Figura 4.8.b se muestra el valor de $CW_{f'}(v)\ \forall\ v \in V$ tras el movimiento $\texttt{INS}(f, 2, B)$. Como se puede observar, el *cutwidth* de los vértices B, C y D, afectados por el movimiento, se ha visto modificado tras la inserción del vértice B en la posición 2, en concreto, en la ordenación f, $CW_f(B) = 4$, $CW_f(C) = 7$ y $CW_f(D) = 7$, mientras que en la ordenación f', $CW_{f'}(B) = 6$, $CW_{f'}(C) = 6$ y $CW_{f'}(D) = 4$. A pesar de que el *cutwidth* del vértice B se ha visto incrementado, el *cutwidth* de los vértices C y D ha decrecido. Además, el *cutwidth* de dichos vértices en la ordenación f determina el *cutwidth* del grafo ($CW_f(G)$) para esta ordenación, por lo que el valor de la función objetivo, también ha decrecido. Es importante destacar que el *cutwidth* de los vértices A, E y F no se ha visto modificado tras la inserción.

Revaluación de una solución tras un movimiento

En una implementación directa, la complejidad de computar el *cutwidth* de cada vértice, $CW_f(v)$, en un grafo G con n vértices y m aristas es $\Theta(m)$ dado que el método debe recorrer todas las aristas del grafo. Adicionalmente, computar el valor de la función objetivo, $CW_f(G)$, es $\Theta(n)$ dado que requiere calcular el máximo $CW_f(v)$ de todos los vértices de G. No obstante, tal y como se ha observado anteriormente, el *cutwidth* de algunos vértices no cambia cuando se realiza un movimiento de inserción, lo que permite ahorrar tiempo en la revaluación de la nueva ordenación.

En el ejemplo de la Figura 4.8.a se tiene la ordenación $f = (A, C, D, B, E, F)$ sobre

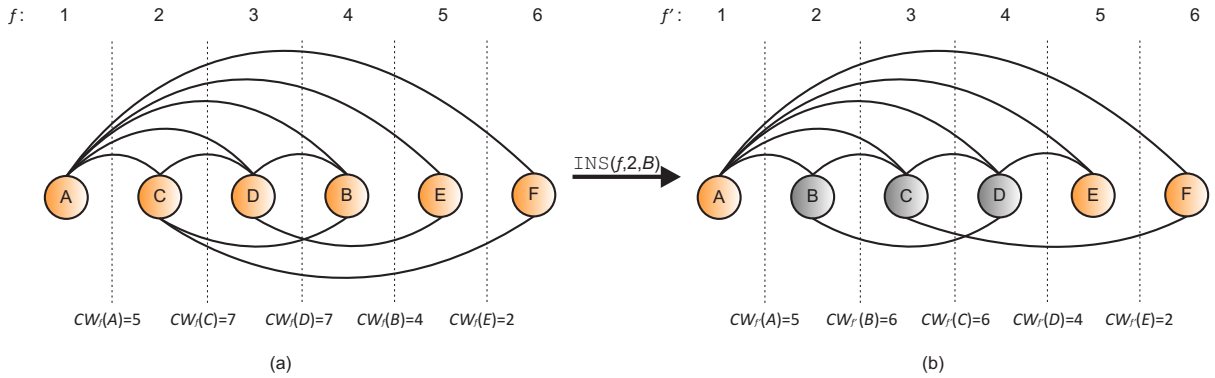


Figura 4.8: (a) Ordenación f de los vértices un grafo G. (b) Ordenación f' tras el movimiento $\texttt{INS}(f, 2, B)$.

la cual se lleva a cabo un movimiento de inserción $\texttt{INS}(f, 2, B)$, obteniendo la ordenación $f' = (A, B, C, D, E, F)$ mostrada en la Figura 4.8.b. Tras realizar este movimiento se puede observar que el *cutwidth* de los vértices A, E y F no se ha modificado, por lo que no es necesario volver a calcularlo.

Cuando se realiza un movimiento $\texttt{INS}(f, j, v)$ donde, por ejemplo $j < f(v)$, para cualquier vértice w tal que $f(w) < j$ o $f(w) > f(v)$, su *cutwidth* no varía. Por el contrario, aquellos vértices situados entre las posiciones j y $f(v)$ cambian su valor del *cutwidth*. Sea $N_i^L(w)$ el conjunto de vértices adyacentes a w situados en cualquier posición previa o igual a i y sea $N_i^R(w)$ el conjunto de vértices adyacentes a w situados en cualquier posición posterior a i. En términos matemáticos:

$$N_i^L(w) = \{u \in V,\ (u, w) \in E\ /\ f(u) \leq i\}$$
$$N_i^R(w) = \{u \in V,\ (u, w) \in E\ /\ f(u) > i\}$$

Por ejemplo, en la Figura 4.8.a el conjunto $N_3^L(D) = \{A, C\}$, ya que ambos vértices son adyacentes a D y están situados en una posición anterior a la posición 3. Análogamente $N_3^R(D) = \{B, E\}$ ya que B y E son vértices adyacentes a D situados a la derecha de dicha posición.

Sea v un vértice con el que se va a realizar un movimiento de inserción en la posición j, $\texttt{INS}(f, j, v)$, y w un vértice tal que $j \leq f(w) < f(v)$, o bien $j \geq f(w) > f(v)$. Es decir, el vértice w se encuentra situado entre la posición del vértice v en la ordenación f y la posición destino del movimiento de inserción. Suponiendo que $f(w) = i$, se podrían definir los conjuntos $N_i^L(v)$ y $N_i^R(v)$ como aquéllos que contienen los vértices adyacentes a v que están situados a la izquierda y derecha, respectivamente, de la posición i en la ordenación f. Dichos conjuntos permiten calcular la diferencia existente entre el $CW_{f'}(w)$ y $CW_f(w)$ empleando la siguiente expresión:

$$CW_{f'}(w) - CW_f(w) = \begin{cases} \mid N_i^L(v) \mid - \mid N_i^R(v) \mid, & si\ j \leq f(w) < f(v) \\ \mid N_i^R(v) \mid - \mid N_i^L(v) \mid, & si\ j \geq f(w) > f(v) \end{cases}$$

donde una diferencia negativa indica que el *cutwidth* del vértice evaluado w ha decrecido tras el movimiento, una diferencia igual a cero indica que no ha variado y una diferencia positiva indica que ha aumentado.

Empleando la expresión anterior, se puede actualizar de manera incremental el *cutwidth* de los vértices que se ven afectados por un movimiento de inserción, lo que reduce el tiempo de cómputo. Además, para calcular el valor de la función objetivo, se almacenan en un conjunto todos los vértices w que satisfacen $CW_f(w) = CW_f(G)$. Si después de realizar un movimiento, este conjunto no es vacío, la función objetivo no cambia. De otro modo, sería necesario actualizar $CW_f(G)$ recorriendo todos los vértices del grafo y seleccionando el valor máximo de entre ellos. En este caso, la actualización de $CW_f(G)$ sería $\mathrm{O}(n)$ en lugar de $\Theta(n)$.

4.4.2. Procedimiento de búsqueda

Cuando el objetivo de un problema de optimización consiste en minimizar un valor máximo, como es el caso del CMP, es común que haya muchas soluciones con el mismo valor de la función objetivo. Se podría decir, de manera coloquial, que el espacio de soluciones presenta un horizonte plano [147, 155]. Esto supone un problema para muchos procedimientos de búsqueda local basados en movimientos, debido a que tras realizar un movimiento, es común que el valor de la función objetivo no cambie, pese a que la estructura de la nueva solución sea distinta.

En el caso particular del CMP, para una ordenación f de los vértices de un grafo que represente una solución al problema, puede haber múltiples vértices cuyo *cutwidth* sea igual al valor de la función objetivo, $CW_f(G)$, para dicha ordenación. En esta situación, realizar un movimiento que decremente el *cutwidth* de un vértice particular u, $CW_f(u)$, no implica necesariamente decrementar también el valor de $CW_f(G)$.

Otra particularidad de este problema es que los vértices con un *cutwidth* próximo a $CW_f(G)$, que no determinan el valor de la función objetivo en la solución actual, son susceptibles a hacerlo en las próximas iteraciones.

Considerando las premisas anteriores, se ha modificado el concepto de movimiento de mejora y, siguiendo la estrategia introducida en [147], se ha definido un conjunto de Vértices Críticos (CV) tal y como se describe a continuación.

Lista de vértices críticos

El conjunto CV está compuesto por aquellos vértices cuyo valor del *cutwidth* es igual o próximo al *cutwidth* del grafo. Estos vértices determinan el valor de la función objetivo en la

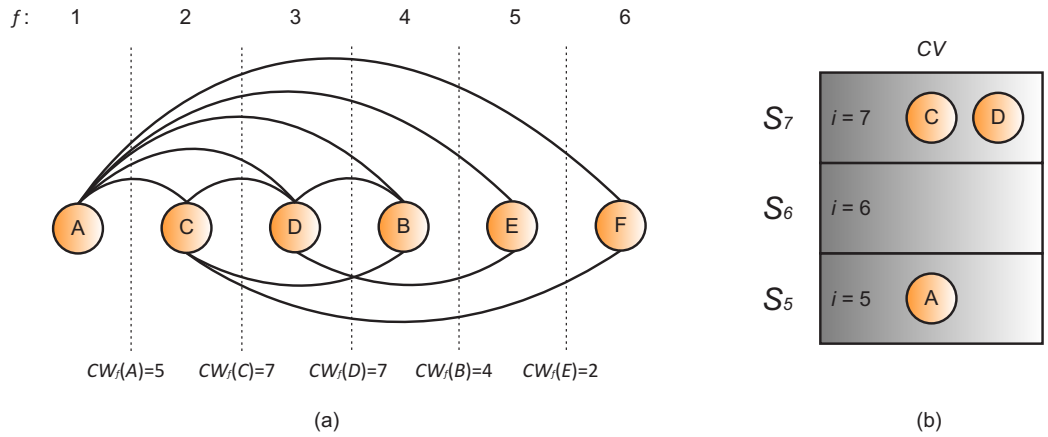


Figura 4.9: (a) Ordenación f de los vértices un grafo G. (b) Lista de vértices críticos (CV) de la ordenación f con un $CW_f(v) \geq 5$.

presente iteración o lo harán, con una alta probabilidad, en próximas iteraciones. El conjunto CV está dividido en subconjuntos $S_i = \{v \in V \,/\, CW_f(v) = i\}$ donde S_i representa al subconjunto de vértices con un valor del *cutwidth* igual a i. En términos matemáticos:

$$CV = \bigcup_{i \geq \lceil \beta \cdot CW_f(G) \rceil} S_i$$

donde β ($0 \leq \beta \leq 1$) es un porcentaje del valor de la función objetivo actual $CW_f(G)$, que determina el valor del umbral para decidir los vértices que forman parte de CV. Este umbral se calcula como $\lceil \beta \cdot CW_f(G) \rceil$. En el Capítulo 5 se estudia la influencia de β en el proceso de búsqueda.

En la Figura 4.9.a se muestra una ordenación de los vértices de un grafo G y el valor $CW_f(v)$ para aquellos vértices $v \in V$ que ocupan una posición evaluable. En la Figura 4.9.b se muestra el conjunto CV considerando que $\beta = 0,7$. Como se puede observar en dicha figura, existen tres vértices $v \in V$ con un $\mathrm{CW}_f(v) \geq \lceil 0,7 \cdot 7 \rceil = 5$. En concreto, los vértices C y D forman el subconjunto S_7 ($S_7 = \{C, D\}$), el conjunto S_6 está vacío, y el vértice A forma el subconjunto S_5 ($S_5 = \{A\}$).

Posiciones candidatas para los vértices críticos

Una vez construido el conjunto de vértices críticos, los conjuntos S_i son recorridos en orden descendente de i, comenzando con $i = CW_f(G)$. En cada iteración, se selecciona un vértice $v \in S_i$ y se evalúa el movimiento $\texttt{INS}(f, pos, v)$ donde la posición pos en la que se insertará el vértice v se calcula como la mediana de las posiciones de sus vértices adyacentes. Situar un vértice en la mediana de las posiciones de sus adyacentes garantiza que el número de adyacentes a cada lado es el mismo o, como máximo, difiere en una unidad si el número de adyacentes es

impar. Esto garantiza que el *cutwidth* generado considerando únicamente las aristas de dicho vértice, sea lo menor posible, lo que puede ayudar a su vez, a reducir el *cutwidth* del grafo.

Si el resultado del movimiento es una mejora, entonces éste se mantiene, siguiendo una estrategia *first improvement*. En caso contrario, se consideran los movimientos $\texttt{INS}(f, j, v)$ con $j \in [pos - w, pos + w]$, donde w es un parámetro de búsqueda establecido como un porcentaje del número de vértices del grafo. Nuevamente, se acepta el primer movimiento que conduzca a una mejora de la solución.

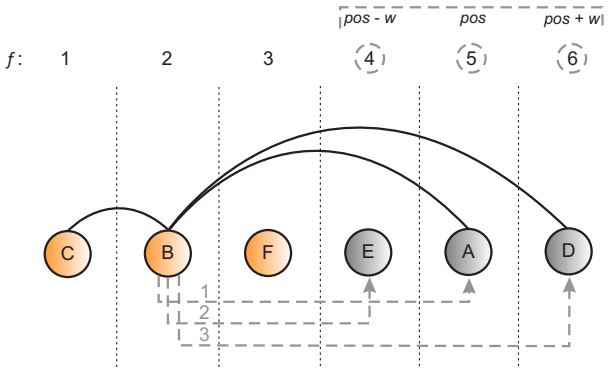


Figura 4.10: Representación de los vértices adyacentes a $B \in CV$ en la ordenación f y de las posiciones candidatas en las que probar la inserción de B.

En la Figura 4.10 se muestra una ordenación f de los vértices de un grafo G, donde únicamente se han representado las aristas que unen al vértice $B \in CV$ con cualquier otro vértice de G. Como se muestra en dicha figura, el vértice B tiene como adyacentes a los vértices C, A y D, que ocupan las posiciones 1, 5 y 6 respectivamente en la ordenación f. La mediana de las posiciones de los adyacentes de C es la posición 5. Supóngase también que se ha establecido un porcentaje del 10 % del número de vértices del grafo, para determinar el máximo número de posiciones en las que insertar cada vértice de CV. Por lo tanto, en el ejemplo de la Figura 4.10, $pos = 5$ y $w = \lceil 6 \cdot 0{,}1 \rceil = 1$, luego las posiciones candidatas en las que insertar el vértice B son las posiciones 4, 5 y 6. En dicha figura se muestra además, con una flecha y un número identificativo, el orden en el que las posiciones candidatas son evaluadas. Destacar que el orden en el que se exploran dichas posiciones es importante, ya que se sigue una estrategia tipo *first improvement*. Inicialmente se comenzaría insertando B en la posición 5, posición de la mediana de las etiquetas de sus adyacentes. A continuación, se explorarían las posiciones más próximas a la mediana aún sin explorar. Para ello, se inserta el vértice candidato, de manera alternativa, a izquierda (posición 4) y a derecha (posición 6) de la mediana. El proceso de exploración con el vértice considerado finaliza cuando se ha realizado un movimiento de mejora, o bien cuando se han explorado todas las posiciones candidatas y ninguna ha producido una mejora.

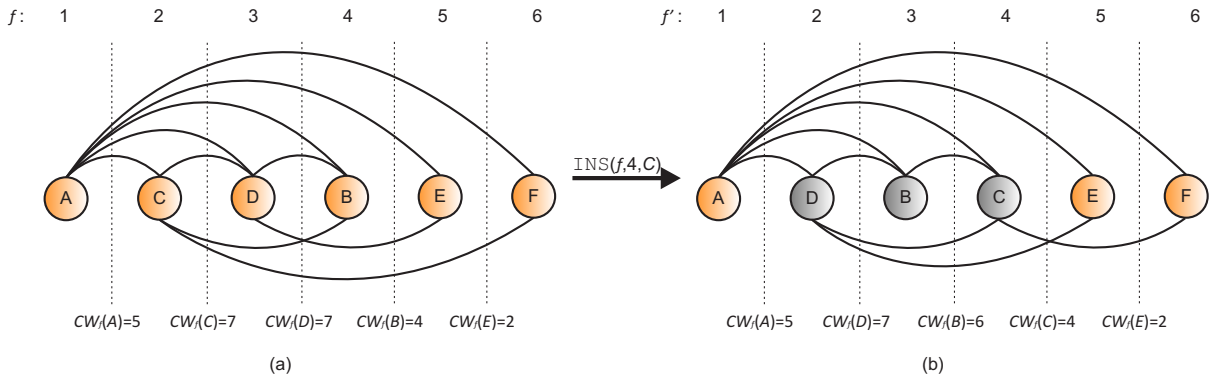


Figura 4.11: (a) Ordenación f de los vértices un grafo G. (b) Ordenación f' obtenida como resultado del movimiento $\texttt{INS}(f, 4, C)$.

Movimientos de mejora en el proceso de búsqueda

Con el objetivo de paliar la falta de información aportada por la escasa variación del valor de la función objetivo tras un movimiento, en esta Tesis Doctoral se ha extendido el concepto de movimiento de mejora. Dado un vértice v, se considera que un movimiento mejora la solución actual si es capaz de reducir la cardinalidad de cualquier conjunto $S_i \subseteq CV$ con $i \geq CW_f(v)$. Es decir, un movimiento de mejora no se limita únicamente a una mejora en el valor de la función objetivo, ya que esto restringiría en exceso la búsqueda, sino que se extiende a una mejora en el *cutwidth* de todo vértice con un *cutwidth* mayor o igual que el evaluado.

En el ejemplo de la Figura 4.11.a (cuyo conjunto CV es igual que el de la Figura 4.9.b) la exploración comenzaría realizando movimientos de inserción con el vértice C, que pertenece al conjunto de vértices con *cutwidth* igual al *cutwidth* del grafo. Los adyacentes de dicho vértice, A, D, B y F, ocupan las posiciones 1, 3, 4 y 6 respectivamente. La primera posición en la que se insertaría el vértice B sería, por lo tanto, la posición 4, por ser la posición situada en la mediana de las etiquetas de sus adyacentes. Cabe destacar que cuando el número de adyacentes es par, es necesario decidir cuál de las dos posiciones (3 y 4, en el ejemplo) se toma como mediana. En este caso, por convenio, se ha tomado la decisión de comenzar por la posición mayor de las dos consideradas. En la Figura 4.11.b se muestra el resultado de realizar el movimiento $\texttt{INS}(f, 4, C)$ sobre la ordenación de la Figura 4.11.a. Tras el movimiento de inserción, en la nueva ordenación, el conjunto S_7 estaría formado únicamente por un vértice ($S_7 = \{D\}$), por lo tanto, pese a que el $CW_{f'}(G) = CW_f(G)$, la ordenación f' ha reducido la cardinalidad de un subconjunto $S_i \subseteq CV$, que representa un *cutwidth* igual o mayor que el del vértice evaluado. Por lo tanto, el movimiento se considera un movimiento de mejora y, al seguir una estrategia *first improvement*, éste se mantiene.

Con el objetivo de mejorar el rendimiento del proceso de búsqueda y tal y como se sugiere en [73], la lista CV no se actualiza en cada iteración del algoritmo, sino después de que todos

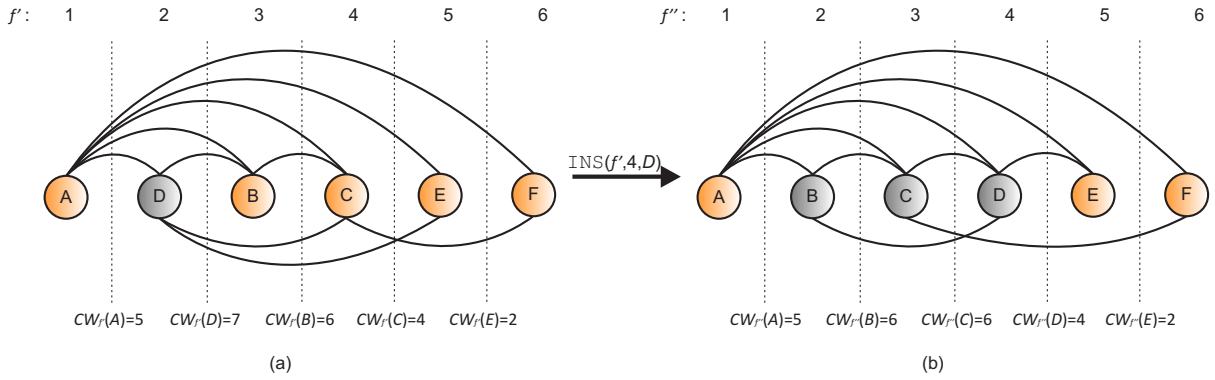


Figura 4.12: (a) Ordenación f' de los vértices un grafo G. (b) Ordenación f'' obtenida como resultado del movimiento $\texttt{INS}(f', 4, D)$.

los vértices en CV hayan sido examinados. Esta estrategia es especialmente útil cuando las actualizaciones provocadas como consecuencia de un movimiento son costosas de recalcular desde el punto de vista computacional.

En la siguiente iteración del algoritmo, se prueban movimientos de inserción con el vértice D. Nuevamente, se calcularía la mediana de las posiciones de sus adyacentes y se comenzaría por esa posición. En concreto, $f'(A) = 1$, $f'(B) = 3$, $f'(C) = 4$ y $f'(E) = 5$, por lo que se probaría la inserción del vértice D en la posición 4 ($\texttt{INS}(f', 4, D)$) obteniendo una ordenación como la mostrada en la Figura 4.12.b. Nuevamente, esta inserción es un movimiento de mejora, ya que se reduce la cardinalidad del conjunto S_7. Además, esta mejora supone una mejora en la función objetivo que pasa de $CW_{f'}(G) = 7$ a $CW_{f''}(G) = 6$.

Por último, antes de actualizar la lista CV se realizarían movimientos de inserción con el vértice A, que pertenece al subconjunto S_5 del conjunto CV. En este caso, el vértice A es adyacente a los vértices B, C, D, E y F que ocupan las posiciones 2, 3, 4, 5 y 6 respectivamente. Nuevamente, el proceso comenzaría insertando el vértice A en la posición 4 de la ordenación f'' obteniendo una solución como la mostrada en la Figura 4.13.b.

Al igual que en las etapas anteriores, dicho movimiento ha reducido la cardinalidad de algún conjunto S_i con $i \geq CW_{f''}(A)$. En concreto, el conjunto S_6 ha pasado a tener de dos elementos, a no tener ninguno. Además, esta reducción ha supuesto una mejora en el valor de la función objetivo.

Una vez que para cada vértice de CV se han examinado todos los movimientos definidos según las premisas anteriores, si alguno de ellos hubiera resultado en un movimiento de mejora, se actualizaría la lista CV a partir de la última solución encontrada y se repetiría el proceso de búsqueda.

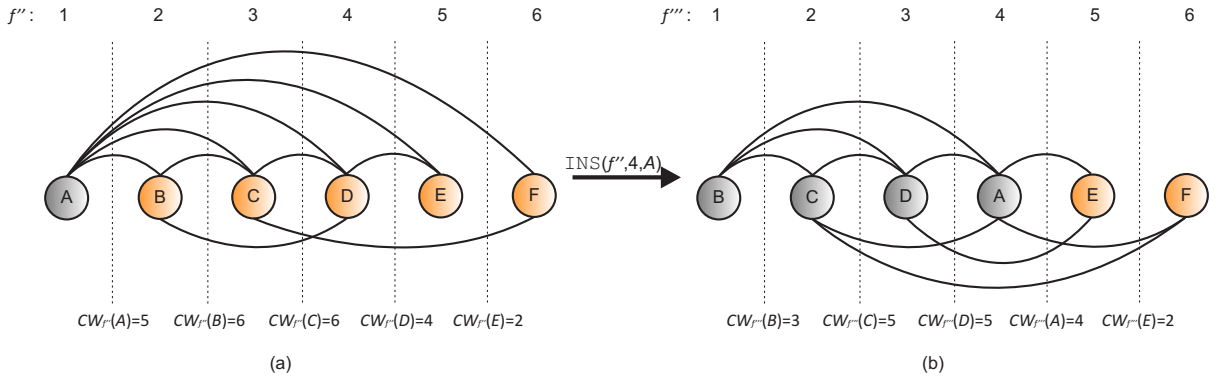


Figura 4.13: (a) Ordenación f'' de los vértices un grafo G. (b) Ordenación f''' obtenida como resultado del movimiento $\texttt{INS}(f'', 4, A)$.

En el ejemplo de la Figura 4.14.a se muestra la última solución obtenida en el ejemplo anterior (f''') tras examinar el primer conjunto CV. A partir de dicha ordenación, se construye un nuevo conjunto de vértices críticos CV', representado en la Figura 4.14.b, donde el $CW_f(G) = 5$, luego para un $\beta = 0{,}7$, entrarían en CV' aquellos vértices $v \in V \,/\, CW_{f'''}(v) \geq \lceil 0{,}7 \cdot 5 \rceil = 4$. Este proceso se repite hasta que no se encuentra ningún movimiento de mejora para algún vértice en CV, momento en el que el proceso de búsqueda local habría finalizado.

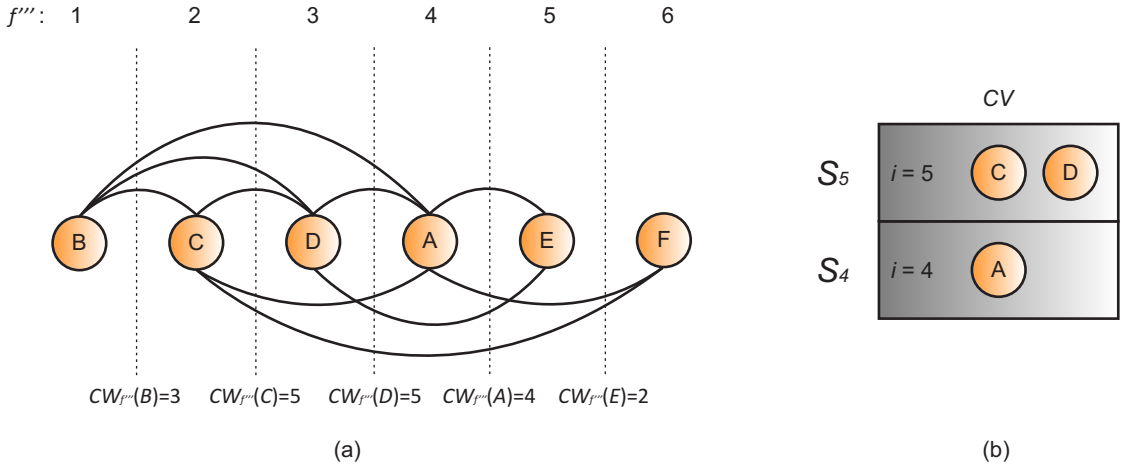


Figura 4.14: (a) Ordenación f''' de los vértices un grafo G. (b) Lista de vértices críticos (CV) de la ordenación f''' con un $CW_f'''(v) \geq 4$.

4.5. Búsqueda Dispersa

En esta Tesis Doctoral se propone la creación de un algoritmo heurístico basado en Búsqueda Dispersa (SS, del inglés *Scatter Search*) para la obtención de soluciones aproximadas al CMP. Para ello, se ha empleado el diseño estándar de SS [100] con la excepción del método de

actualización del denominado «Conjunto de Referencia» (*RefSet*, del inglés *Reference Set*) y la posibilidad de regeneración del mismo.

En el Capítulo 1 se realiza una introducción a la metaheurística SS desde una perspectiva general. A continuación, se detalla la adaptación de cada una de las etapas de SS (generación de soluciones diversas, mejora de las soluciones, creación del *RefSet*, generación de subconjuntos, combinación de soluciones y actualización del *RefSet*) a la resolución del CMP:

- **Método de generación de soluciones diversas**: como se indica en el Capítulo 1, SS parte de un conjunto de soluciones que pueden ser generadas empleando una heurística constructiva. Dichas soluciones deben tener una buena calidad, pero también ser suficientemente diversas, permitiendo a la búsqueda local y al método de combinación alcanzar diferentes óptimos locales [100]. El procedimiento de generación de soluciones diversas se utiliza tanto en la creación del conjunto inicial de soluciones, que se denominará en lo que sigue «Conjunto Diverso», como en la regeneración del mismo, si todas las soluciones iniciales hubieran sido examinadas y los parámetros de exploración así lo exigieran. En la Sección 4.3 se proponen cuatro procedimientos constructivos basados en la metodología GRASP destinados a la generación de soluciones para el CMP. Cada uno de dichos procedimientos puede ser adecuado para generar el «Conjunto Diverso». La determinación de cuál de los algoritmos constructivos emplear en el esquema definitivo de SS se lleva a cabo en el Capítulo 5. Para ello, se realiza un experimento preliminar analizando las soluciones producidas por los distintos procedimientos constructivos y, empleando tanto criterios de calidad como de diversidad, se escoge la heurística constructiva más adecuada.

- **Método de mejora**: el esquema de SS incluye la utilización de un método de mejora en diversas fases del algoritmo, con el objetivo de encontrar soluciones de mejor calidad en la vecindad de soluciones previamente creadas. Concretamente, el método de mejora se emplea sobre las soluciones construidas por el método de generación de soluciones diversas. Es decir, sirve para actualizar las soluciones que forman el «Conjunto Diverso», en el caso de que el método de mejora sea capaz de encontrar una solución de mejor calidad a partir de alguna de ellas. Además, también se utiliza como una fase de proceso posterior a la combinación de soluciones, sobre las soluciones producidas por el método de combinación. En esta Tesis Doctoral se propone la utilización de la búsqueda local basada en inserciones, descrita en la Sección 4.4, como método de mejora dentro del esquema algorítmico de SS.

- **Método de creación del *RefSet***: una vez se dispone de un «Conjunto Diverso», construido empleando el método de generación de soluciones diversas, y posteriormente mejorado empleando la búsqueda local anteriormente descrita, se construye el *RefSet*. Inicialmente el *RefSet*, con $|RefSet| = b$, es un subconjunto del «Conjunto Diverso»

formado de la siguiente manera: se seleccionan las $b/2$ mejores soluciones del «Conjunto Diverso» y se añaden al *RefSet*. Las $b/2$ soluciones restantes se añaden al *RefSet* empleando criterios de diversidad. Es decir, parte de las soluciones son escogidas por su calidad y el resto por ser diferentes respecto a las primeras.

Para determinar qué soluciones del «Conjunto Diverso» son distintas frente a las ya añadidas al *RefSet* se emplea una medida de distancia entre soluciones. En concreto, la medida de distancia empleada en esta Tesis Doctoral, para determinar qué soluciones se añaden al *RefSet* por diversidad, se describe en la Sección 4.5.1. Las soluciones diversas son seleccionadas de una en una, seleccionando en cada iteración la que tenga una mayor distancia al conjunto, tal y como se describe en 4.5.1. Una vez seleccionada una solución diversa, ésta se añade al $RefSet$ y es considerada parte del conjunto para añadir la siguiente solución diversa. El proceso se repite hasta que se completa el *RefSet*.

En cuanto al tamaño del «Conjunto Diverso» y del *RefSet* se refiere, éste puede ser variable según el problema abordado. Es habitual que el «Conjunto Diverso» tenga un tamaño inicial de 100 soluciones, de las cuales 10 formarán el *RefSet*. El número de soluciones que se incorporan inicialmente al *RefSet* determina el tamaño del mismo para toda la ejecución del algoritmo. El tamaño del *RefSet* puede influir en la calidad de las soluciones obtenidas, por lo que dicho tamaño se puede considerar un parámetro del proceso búsqueda. Por ello, su influencia es evaluada en el Capítulo 5. Una vez definido el tamaño del *RefSet*, es también relevante determinar el número de soluciones escogidas por calidad y el número de soluciones escogidas por diversidad. Una configuración típica selecciona el 50 % de las soluciones por calidad y el 50 % por diversidad. Dado que estos porcentajes pueden ser ajustados en función del problema abordado, nuevamente en el Capítulo 5 se evalúa el impacto de la composición inicial del *RefSet*.

- **Método de generación de subconjuntos**: la metodología SS está basada en la combinación de las soluciones presentes en el *RefSet* para obtener nuevas soluciones al problema. Para ello, se debe determinar cuántas soluciones es necesario combinar para generar una nueva solución. Este número determina el tamaño de los subconjuntos que se deben formar. Una vez establecido este parámetro, se empleará el método de generación de subconjuntos para, como su propio nombre indica, generar todos los subconjuntos de soluciones posibles del tamaño determinado, a partir de las soluciones contenidas en el *RefSet*. En el diseño particular de un algoritmo de SS para el CMP se propone la creación de subconjuntos de 2 soluciones. De este modo, suponiendo un tamaño del *RefSet* de 10 soluciones y que ninguna de ellas hubiera sido explorada todavía, se dispondría de un total de 45 subconjuntos diferentes de 2 soluciones cada uno para realizar combinaciones.

- **Método de combinación**: el método de combinación posibilita la creación de nuevas soluciones a partir de los subconjuntos creados anteriormente, con las soluciones existentes

en el *RefSet*. Esta técnica se fundamenta en la idea de que, una solución obtenida a partir de la selección de atributos presentes en soluciones de calidad, debe generar soluciones con una buena estructura. Además, introducir ciertas soluciones de menor calidad en el *RefSet*, pero diferentes a las primeras, permite diversificar el proceso de búsqueda, explorando así otras regiones del espacio de soluciones. Para llevar a cabo el proceso de generación de una solución a partir de otras soluciones ya existentes, se emplea el denominado método de combinación de soluciones. En concreto, en esta Tesis Doctoral se proponen tres métodos de combinación de soluciones distintos para el CMP. Dichos métodos son descritos de manera detallada en la Sección 4.5.2. Al igual que con los procedimientos constructivos, la selección del método incluido en el esquema definitivo de SS se ha realizado de manera experimental y puede encontrarse en el Capítulo 5.

- **Método de actualización del *RefSet***: como resultado del método de combinación, se obtiene un conjunto de nuevas soluciones. Dichas soluciones son evaluadas una a una para determinar si se incluyen en el $RefSet$, en la siguiente iteración del algoritmo. Para ello, las soluciones en el *RefSet* (de la iteración actual) se ordenan de acuerdo con su calidad, donde la solución x^1 representa a la solución con mayor calidad de las que componen el *RefSet*, y la solución x^b a la solución con menor calidad, en términos del valor la función objetivo. Dada una solución x generada por el método de combinación, se denota como y^x a la solución del *RefSet* más parecida a x, que se determina empleando la medida de distancia descrita en la Sección 4.5.1. La solución x es seleccionada para formar parte del *RefSet* en la siguiente iteración del algoritmo, si se cumple alguno de los dos siguientes criterios:

  - x es mejor que la mejor solución en el *RefSet* (x^1).
  - x es peor que la mejor solución el *RefSet* (x^1), pero es mejor que la peor solución del mismo (x^b) y su distancia con respecto a y^x es mayor que un determinado umbral ($dthresh$) previamente establecido.

  Si se determina que la solución x debe entrar en el *RefSet*, entonces sustituirá a la solución y^x (siempre que esta no sea la mejor del *RefSet*) manteniendo así el tamaño del mismo constante y la máxima diversidad posible.

En Algoritmo 4.5 se muestra una adaptación del pseudocódigo de SS propuesto en [120] de acuerdo con el esquema empleado para la resolución del CMP. Dicho algoritmo recibe los siguientes parámetros de entrada: el tamaño del «Conjunto Diverso» (N); el tamaño del *RefSet* (b); el porcentaje de soluciones que se añaden al *RefSet* por diversidad ($pDivRS$); el tamaño de los subconjuntos a combinar ($tamSC$); el número de regeneraciones del *RefSet* permitidas ($numReg$) y un porcentaje sobre la distancia máxima de una solución del «Conjunto Diverso» al *RefSet* ($pDis$), que interviene en el criterio de aceptación de soluciones en el mismo.

Algoritmo 4.5 Algoritmo de Búsqueda Dispersa adaptado de [120]

```
1: procedimiento BúsquedaDispersa (N, b, pDivRS, tamSC, numReg, pDis)
2: Sea ConjuntoDiverso el conjunto inicial de soluciones diversas;
3: Sea RefSet el conjunto de soluciones de referencia;
4: Inicialmente ConjuntoDiverso = ∅ y RefSet = ∅;
5: ConjuntoDiverso ← ConstruirSoluciones(N);
6: ConjuntoDiverso ← MejorarSoluciones(ConjuntoDiverso);
7: RefSet ← ConstruirRefSet(ConjuntoDiverso, b, q);
8: SolucionNueva ← True;
9: mientras numReg > 0 hacer
10:     dthresh ← calcularUmbral(pDis);
11:     mientras SolucionNueva hacer
12:         x^1 ← mín(RefSet)
13:         x^m ← máx(RefSet)
14:         SolucionNueva ← False;
15:         P ← GenerarGrupos(tamSC);
16:         mientras P ≠ ∅ hacer
17:             p_nueva ← Combinar(par_i ∈ P);
18:             P = P \ par_i;
19:             p_nueva ← MejorarSoluciones(p_nueva);
20:             si p_nueva < x^1 entonces
21:                 ActualizarRefSet(p_nueva);
22:                 SolucionNueva ← True;
23:             si no
24:                 si p_nueva < x^m y CalcularDiversidad(p_nueva) > dthresh entonces
25:                     ActualizarRefSet(p_nueva);
26:                     SolucionNueva ← True;
27:                 fin si;
28:             fin si;
29:         fin mientras;
30:     fin mientras;
31:     numReg ← numReg − 1
32:     si numReg > 0 entonces
33:         RefSet ← RegenerarRefSet(RefSet, ConjuntoDiverso);
34:     fin si;
35: fin mientras;
36: fin BúsquedaDispersa;
```

En los pasos 5 y 6 del pseudocódigo se construye y mejora, respectivamente, el «Conjunto Diverso» formado por un total de N soluciones. En el paso 7 se inicializa el *RefSet* con las $b - \lfloor b \cdot pDivRS \rfloor$ mejores soluciones del «Conjunto Diverso» y las $\lfloor b \cdot pDivRS \rfloor$ soluciones más diferentes a las anteriores. Una vez finalizado el proceso de inicialización, en el paso 10, se calcula el valor del umbral *dthresh* como un porcentaje ($pDis$) de la mayor distancia de una solución del «Conjunto Diverso» al *RefSet*. Entre los pasos 11 y 30 se lleva a cabo el proceso de exploración con las soluciones del *RefSet*. Inicialmente se determina cuál es la solución de mejor calidad y cuál la de peor calidad del *RefSet* respectivamente (pasos 12 y 13) y se generan todos los posibles subconjuntos de tamaño *tamSC* con soluciones del *RefSet* (paso 15). Entre los pasos 16 y 28

se examina si, cada nueva solución producida como combinación de un subconjunto (paso 17) pasa a formar parte del *RefSet* (pasos 20-28). En el caso de que se haya actualizado el *RefSet* tras examinar todas las soluciones combinadas, es decir, si se han añadido nuevas soluciones al mismo, el proceso se exploración se repite formando nuevos subconjuntos con las soluciones no exploradas. Cuando ninguna combinación de las soluciones del *RefSet* sea capaz de producir una solución que pueda entrar en el *RefSet*, el proceso finaliza. En esa situación, si el parámetro *numReg* es mayor que cero se regenera el *RefSet* (pasos 31-33) eliminando un porcentaje de las soluciones de peor calidad del mismo y añadiendo las soluciones más diversas, presentes en el «Conjunto Diverso» y no consideradas anteriormente, respecto de las soluciones que permanecen en el *RefSet*. De este modo, el proceso de exploración comenzaría una vez más, con el nuevo *RefSet*.

Muchas propuestas basadas en SS incluyen variaciones del esquema general que permiten aumentar su rendimiento. Algunas de ellas pueden encontrarse en [74, 75, 120]. En el algoritmo de SS propuesto en esta Tesis Doctoral se han incluido dos variaciones significativas en dicho esquema, que pueden verse reflejadas en el Algoritmo 4.5: el criterio de actualización del $RefSet$ y la posibilidad de regeneración del mismo.

La actualización del conjunto de referencia se ha basado, de manera tradicional, únicamente en criterios de calidad. En este caso, además de la actualización por calidad (cuando la solución combinada es la mejor de todas las encontradas hasta el momento) se ha considerado una actualización basada en la combinación de criterios de calidad y diversidad. Si una solución combinada es mejor que la peor solución del *RefSet*, aunque no sea la mejor de todas, y además es suficientemente diversa, se añade al *RefSet*. Se considera que una solución es suficientemente diversa si la distancia de la misma al *RefSet* es mayor que un umbral (*dthresh*). Este umbral se determina como un porcentaje sobre la distancia máxima de cualquier solución del «Conjunto Diverso», al *RefSet*.

Por otro lado, en cuanto a la terminación del algoritmo se refiere, es habitual determinar que ha finalizado su ejecución cuando no es posible añadir nuevas soluciones combinadas al *RefSet*. No obstante, existe una variante en la que se introduce la posibilidad de regenerar el *RefSet* un número determinado de veces, en el caso de que éste no permita generar nuevas soluciones que cumplan los criterios de entrada en el *RefSet*. En dicha regeneración se elimina un porcentaje de las soluciones de menor calidad y se añaden soluciones del «Conjunto Diverso» que sean diferentes respecto a las soluciones que permanecen en el *RefSet*.

4.5.1. Distancia entre soluciones

En este contexto, las medidas de distancia son funciones utilizadas para determinar las diferencias existentes entre soluciones de un mismo problema de optimización. La distancia entre dos soluciones se representa mediante un valor numérico, que es calculado comparando

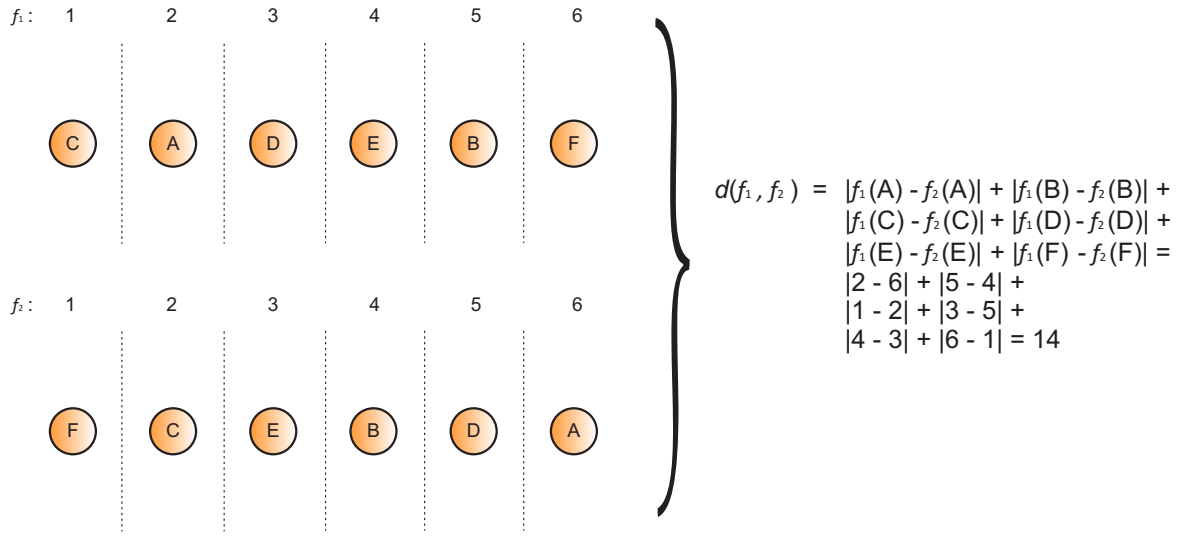


Figura 4.15: Representación de la distancia entre las soluciones f_1 y f_2.

ciertos atributos de cada solución que han sido previamente determinados.

En el esquema de SS la medida de distancia empleada es especialmente relevante. Dicha medida es empleada en el proceso de formación del *RefSet*, así como en su actualización. En concreto, cuando se forma el *RefSet* se escogen parte de las soluciones añadidas según su calidad. El resto, deben ser soluciones diferentes respecto a las primeras. La diversidad de dichas soluciones se mide empleando la medida de distancia. Además, en el diseño particular propuesto en esta Tesis Doctoral, la medida de distancia se emplea también durante el proceso de actualización del *RefSet* tal y como se explicó en la Sección 4.5.

Dado que las soluciones del CMP son ordenaciones de los vértices de un grafo, se propone la utilización de una de las distancias descritas en [100]. En concreto, la distancia entre dos soluciones dadas f_1 y f_2, denotada como $d(f_1, f_2)$, se calcula de la siguiente manera:

$$d(f_1, f_2) = \sum_{v \in V} \mid f_1(v) - f_2(v) \mid$$

En la Figura 4.15 se muestran dos ejemplos de ordenaciones (f_1 y f_2) de los vértices de un grafo G y la distancia entre ellas según la expresión anterior. Esta distancia se calcula de la siguiente manera: $d(f_1, f_2) = \mid f_1(A) - f_2(A) \mid + \mid f_1(B) - f_2(B) \mid + \mid f_1(C) - f_2(C) \mid + \mid f_1(D) - f_2(D) \mid + \mid f_1(E) - f_2(E) \mid + \mid f_1(F) - f_2(F) \mid$.

Típicamente, en SS se calcula la distancia de una solución a un conjunto de soluciones de dos formas distintas. La primera opción se basa en la mínima distancia entre la solución considerada y cualquier solución del conjunto. La segunda opción consiste en calcularla como la suma de las distancias entre dicha solución y cada una de las soluciones del conjunto. Para este problema se ha considerado que la distancia mínima ayuda a obtener soluciones más separadas en el espacio

de soluciones.

4.5.2. Métodos de combinación de soluciones

Como se ha descrito anteriormente, la combinación de soluciones es un aspecto relevante en el éxito de la metodología SS. Los métodos de combinación de soluciones son procedimientos destinados a obtener soluciones nuevas (que se denominarán en lo que sigue «soluciones combinadas») a partir de un conjunto de soluciones ya existentes, denominadas «soluciones de referencia». Existen diferentes métodos de combinación en función del tipo solución que se desee combinar. En particular, cuando la estructura de la solución es una ordenación (como es el caso del CMP) es común utilizar el denominado método de combinación por votos [69]. En él, cada solución de referencia trata de añadir sus características a la solución combinada, votando por ellas en cada iteración del algoritmo. Por lo tanto, se espera que las soluciones combinadas tengan atributos en común con las distintas soluciones de referencia.

En esta Tesis Doctoral se proponen tres métodos de combinación de soluciones para el CMP, que consisten en distintas variantes del método de combinación por votos anteriormente mencionado. Cada uno de los procedimientos propuestos emplea pares de soluciones de referencia, con el objetivo de obtener una solución combinada. Dichos pares de soluciones son subconjuntos formados con el «método de generación de subconjuntos» descrito en la Sección 1.4.2.

Los métodos de combinación propuestos han sido denominados CM1, CM2 y CM3, empleando para ello el acrónimo CM (del inglés *Combination Method*). La diferencia fundamental entre los tres métodos reside en el criterio de selección empleado a la hora de asignar un nuevo vértice, a la siguiente posición libre de la solución combinada. A continuación, se exponen cada uno de los métodos mencionados de manera detallada.

Combinación de soluciones por votos (CM1)

El método de combinación de soluciones denominado CM1 es el método de votos «clásico» introducido en [69]. En él, cada una de las dos soluciones de referencia vota por el elemento de su propia solución que, no estando ya incluido en la solución combinada, ocupa una posición menor en la ordenación. En cada iteración del algoritmo, uno de los dos elementos votados por cada solución de referencia es seleccionado para pasar a formar parte de la solución combinada, asignándole la etiqueta más baja aún disponible. Si la posición en la que están situados cada uno de los elementos votados coincide, se elige aleatoriamente uno de ellos para ser añadido a la solución combinada. En cambio, si las soluciones de referencia votan por elementos que están en distintas posiciones, se añade a la solución combinada el elemento que ocupe una posición menor.

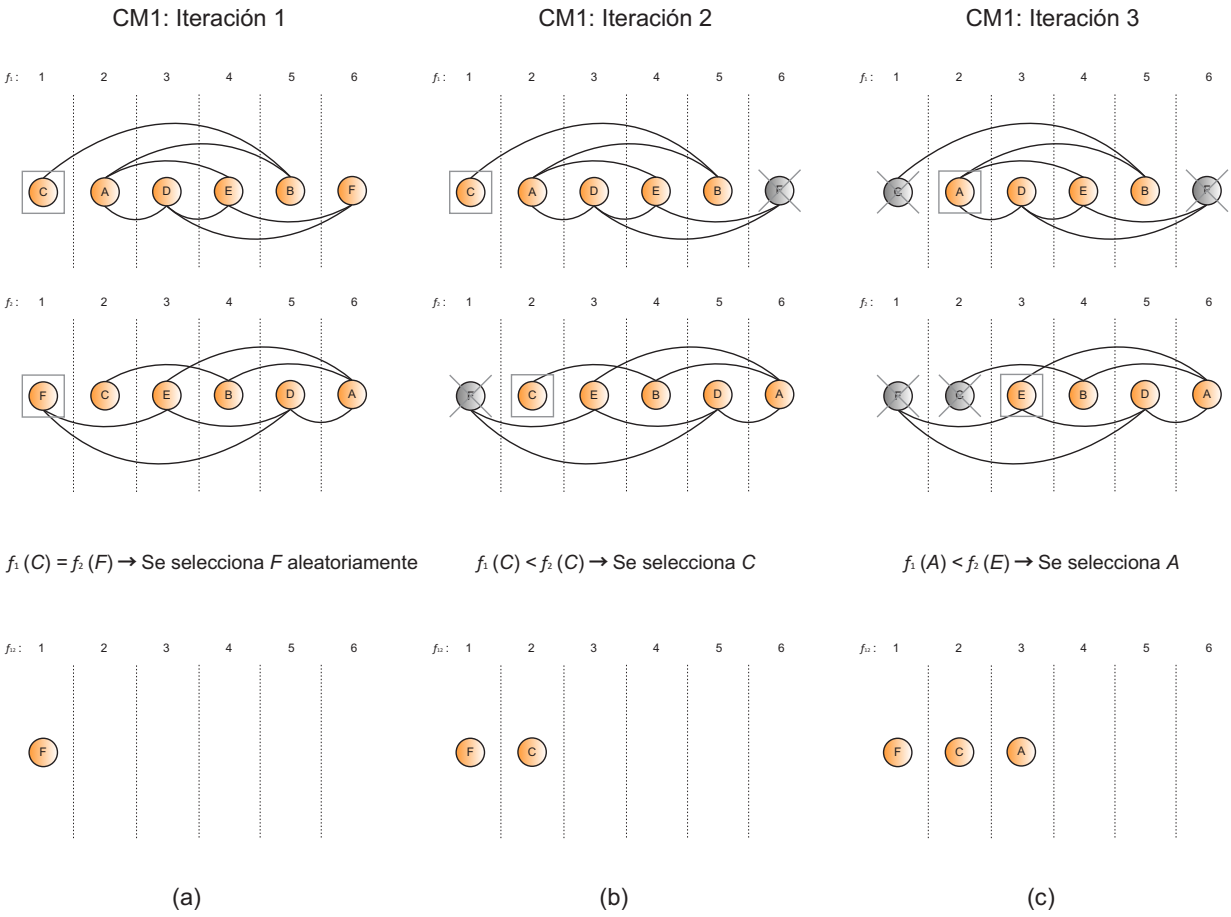


Figura 4.16: Método de combinación CM1 aplicado sobre las soluciones de referencia f_1 y f_2 para obtener la solución combinada f_{12}.

En la Figura 4.16 se representan las tres primeras iteraciones del método de combinación CM1, donde las soluciones f_1 y f_2 de un grafo G son combinadas para obtener la solución f_{12}. La Figura 4.16.a representa el primer paso del algoritmo. En este caso, cada solución de referencia vota por el vértice que ocupa la primera posición en cada una de las ordenaciones. Es decir, f_1 vota por el vértice C y f_2 vota por el vértice F. Dado que el criterio de selección está basado en la posición del mismo, ésta supone una situación de empate. Para romper el empate se selecciona uno de los vértices candidatos de manera aleatoria. Suponiendo que el vértice elegido de manera aleatoria es el vértice F se obtendría la solución ilustrada en la parte inferior izquierda de la figura. Este vértice no volverá a estar disponible para las sucesivas iteraciones del algoritmo. En la segunda iteración, mostrada en la Figura 4.16.b, la solución f_1 vota por el vértice C, que también es votado por la solución f_2.

En esta situación el vértice C es seleccionado para ocupar la siguiente posición disponible en la solución combinada, debido a que ambas soluciones votan por él. En el caso de que fueran distintos, se añadiría el vértice correspondiente de la solución f_1 por ocupar una posición menor en la ordenación. Al igual que ocurre con F, el vértice C no volverá a estar disponible para

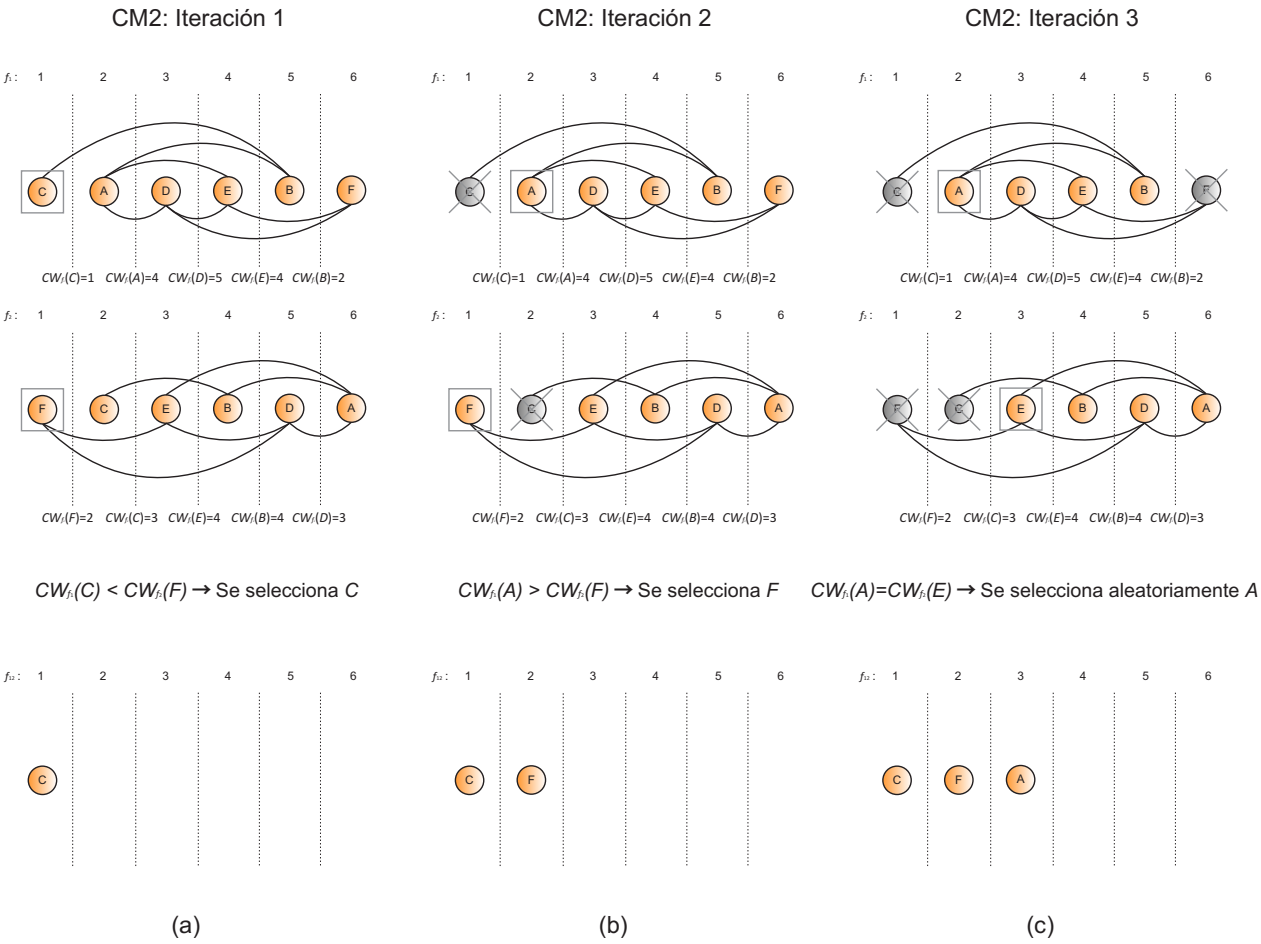


Figura 4.17: Método de combinación CM2 aplicado sobre las soluciones de referencia f_1 y f_2 para obtener la solución combinada f_{12}.

la selección en sucesivas iteraciones. En la Figura 4.16.c se muestra la tercera iteración del algoritmo, donde la solución f_1 vota por el vértice A (con etiqueta 2) y la solución f_2 vota por el vértice E (con etiqueta 3) siendo seleccionado el vértice A. Este procedimiento se repite hasta que se completa la solución combinada.

Combinación de soluciones basada en el *cutwidth* de cada vértice candidato (CM2)

El método de combinación de soluciones CM2 está basado en el *cutwidth* del vértice votado por cada solución de referencia. Al igual que en CM1, en cada iteración, se añade un nuevo elemento a la solución combinada, seleccionándolo de entre los elementos votados y asignándole la etiqueta más baja disponible en la solución combinada. Nuevamente, cada solución de referencia vota por el elemento que ocupe una posición menor en la ordenación y que no haya sido previamente añadido a la solución combinada. De entre los elementos votados se añadirá aquél que tenga un menor valor del *cutwidth*. En caso de empate se selecciona uno de ellos de manera aleatoria.

En la Figura 4.17 se representan las tres primeras iteraciones del método de combinación CM2, donde las soluciones f_1 y f_2 de un grafo G son combinadas para obtener la solución f_{12}. La Figura 4.17.a representa el primer paso del algoritmo. En este ejemplo, cada solución de referencia vota por el vértice que ocupa la primera posición en cada una de las ordenaciones. En concreto, la solución f_1 vota por el vértice C, con un $CW_{f_1}(C) = 1$ y la solución f_2 vota por el vértice F, con un $CW_{f_2}(F) = 2$. Dado que el criterio seguido para seleccionar el siguiente vértice está basado en el valor del *cutwidth* de los vértices candidatos, se selecciona el vértice C. La solución obtenida se ilustra en la parte inferior izquierda de la figura. Este vértice no volverá a estar disponible para las sucesivas iteraciones del algoritmo. En la segunda iteración, mostrada en la Figura 4.17.b, la solución f_1 vota por el vértice A, con un $CW_{f_1}(A) = 4$ mientras que la solución f_2 vota por el vértice F, con un $CW_{f_2}(F) = 2$. En esta ocasión el vértice F es seleccionado para ocupar la siguiente posición disponible en la solución combinada, debido a que su *cutwidth* es menor que el del vértice A. Al igual que ocurre con C, el vértice F no volverá a estar disponible para la selección en sucesivas iteraciones. En la Figura 4.17.c se muestra la tercera iteración del algoritmo, donde la solución f_1 vota por el vértice A y la solución f_2 vota por el vértice E, ambos con *cutwidth* 4. En esta situación el desempate se produce escogiendo entre los vértices A y E de manera aleatoria. Suponiendo que es seleccionado el vértice A, se obtendría la solución mostrada en la parte inferior derecha de la figura. Este procedimiento se repite hasta que se completa la solución combinada.

Combinación de soluciones basada en la función objetivo global (CM3)

El último método de combinación de soluciones propuesto, CM3, está basado en la función objetivo de la solución combinada. En cada iteración, CM3 añade a la solución combinada el elemento votado que menos incremente la función objetivo, de la solución parcial que se está construyendo, asignándole la etiqueta más baja disponible. Al igual que CM1 y CM2, cada solución de referencia vota por el elemento de su propia solución que se encuentre en una posición menor y que no haya sido previamente añadido a la solución combinada. De entre los elementos votados, el elemento elegido será aquél que aporte menos a la función objetivo de la solución combinada. En caso de que ambos elementos tengan la misma aportación se seleccionaría uno de ellos de manera aleatoria.

En la Figura 4.18 se representan las tres primeras iteraciones del método de combinación CM3, donde las soluciones f_1 y f_2 de un grafo G son combinadas para obtener la solución f_{12}. La Figura 4.18.a representa el primer paso del algoritmo. En este ejemplo, cada solución de referencia vota por el vértice que ocupa la primera posición en cada una de las ordenaciones. En concreto, la solución f_1 vota por el vértice C, que generaría un *cutwidth* en la solución combinada de $CW_{f_{12}}(C) = 1$ y la solución f_2 vota por el vértice F, el cual generaría un *cutwidth* en la solución combinada de $CW_{f_{12}}(F) = 2$. Dado que el criterio seguido para seleccionar el

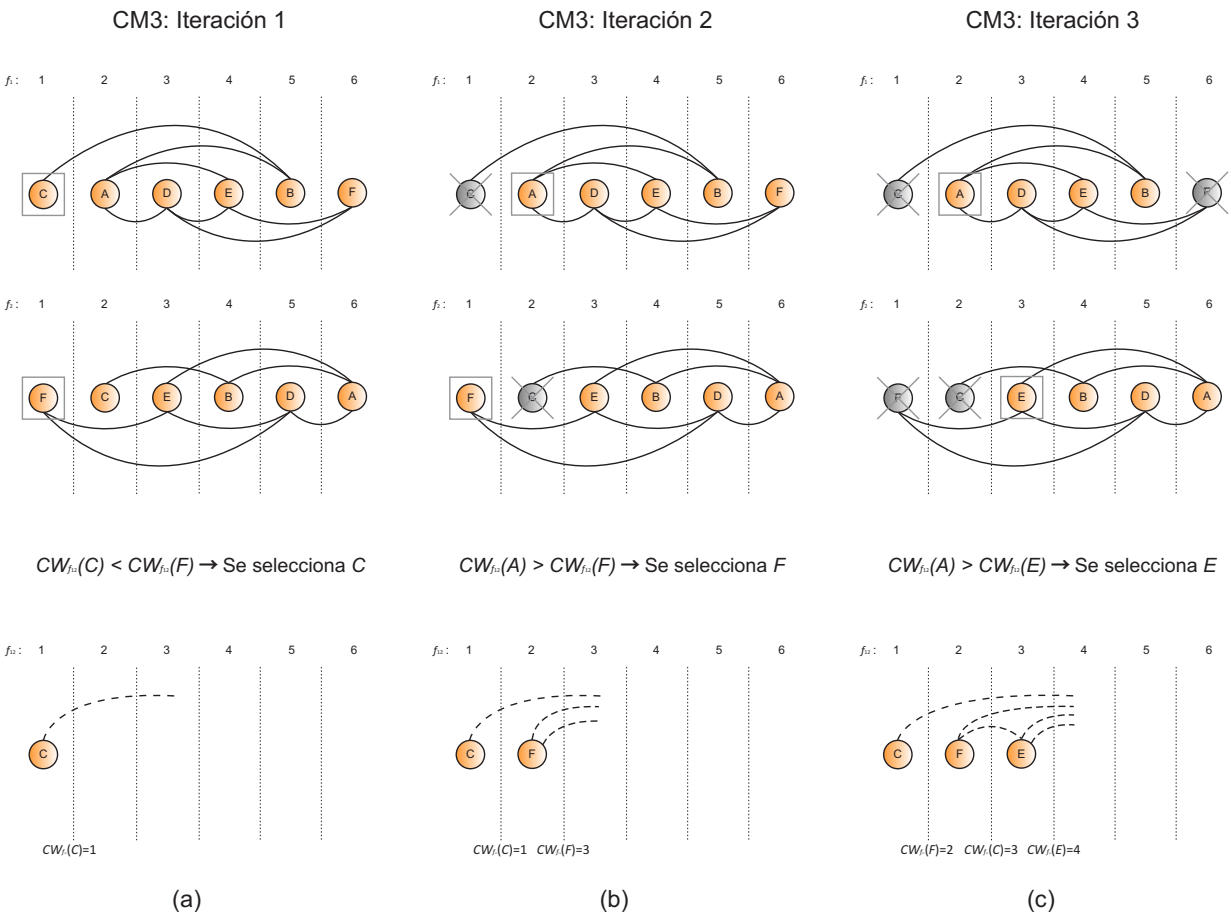


Figura 4.18: Método de combinación CM3 aplicado sobre las soluciones de referencia f_1 y f_2 para obtener la solución combinada f_{12}.

siguiente vértice está basado en *cutwidth* de la solución combinada, se selecciona el vértice C. La solución obtenida se ilustrada en la parte inferior izquierda de la figura. Este vértice no volverá a estar disponible para las sucesivas iteraciones del algoritmo. En la segunda iteración, mostrada en la Figura 4.18.b, la solución f_1 vota por el vértice A, que generaría un *cutwidth* en la solución combinada de $CW_{f_{12}}(A) = 4$ mientras que la solución f_2 vota por el vértice F, que generaría un *cutwidth* en la solución combinada de $CW_{f_{12}}(F) = 3$. En esta ocasión el vértice F es seleccionado para ocupar la siguiente posición disponible en la solución combinada, debido a que el *cutwidth* generado es menor que el que generaría el vértice A. Al igual que ocurre con C, el vértice F no volverá a estar disponible para la selección en sucesivas iteraciones.

En la Figura 4.18.c se muestra la tercera iteración del algoritmo, donde la solución f_1 vota por el vértice A y la solución f_2 vota por el vértice E. En esta situación el vértice A generaría un $CW_{f_{12}}(A) = 5$, mientras que el vértice E generaría un $CW_{f_{12}}(E) = 4$, por lo que se escoge el vértice E. La solución obtenida tras seleccionar el vértice E se muestra en la parte inferior derecha de la figura. Al igual que CM1 y CM2, en caso de empate en el valor del *cutwidth* generado en la solución combinada, el siguiente vértice a añadir a la solución se escogería de

manera aleatoria. Este procedimiento se repite hasta que se completa la solución combinada.

Capítulo 5

Resultados experimentales

El proceso de experimentación permite validar los algoritmos propuestos mediante la comparación de los resultados obtenidos por éstos con los alcanzados por los métodos del estado del arte. En este capítulo, se presentan los resultados experimentales obtenidos en esta Tesis Doctoral para la resolución del «problema de la minimización de la anchura de corte en ordenaciones lineales». En él, se lleva a cabo una extensa experimentación, tanto con los algoritmos exactos propuestos en el Capítulo 3, como con los algoritmos heurísticos propuestos en el Capítulo 4, sobre diversos conjuntos de instancias de referencia.

5.1. Introducción

La experimentación supone un proceso común en ciencia y tecnología para el estudio de algún fenómeno determinado, que permite extraer conclusiones sobre el mismo. En el área de optimización, la experimentación permite realizar comparativas entre diferentes propuestas algorítmicas para la resolución de un mismo problema. Además, también es de utilidad en la validación y depuración del código fuente y en el proceso de ajuste de parámetros de los algoritmos.

Algunos aspectos determinantes en el proceso de experimentación son el conjunto de instancias sobre el que llevar a cabo la experimentación, las métricas de calidad empleadas para comparar las distintas propuestas y las características del ordenador donde se realiza. En las secciones 5.1.1, 5.1.2 y 5.1.3 se repasan algunos de estos aspectos.

5.1.1. Conjuntos de instancias de referencia

Para evaluar el rendimiento de los algoritmos propuestos se han empleado tres tipos de instancias diferentes, presentes en el estado del arte. El primero («Small») fue introducido en [119]; el segundo («Grid») se describe en [159]; y el tercero («Harwell-Boeing») es un

subconjunto del *Matrix Market library*[1]. Todas estas instancias se encuentran disponibles en `http://heur.uv.es/optsicom/cutwidth`. De cada uno de los tipos de instancias anteriormente mencionados, se han creado dos conjuntos diferentes que son empleados en la experimentación de los algoritmos exactos y heurísticos respectivamente. Cada uno de dichos conjuntos es descrito a continuación:

- **Small:** este conjunto de datos está formado por un total de 84 grafos establecidos en el contexto del «*Bandwidth Reduction Problem*» [119]. El número de vértices de dichos grafos oscila entre 16 y 24 y el número de aristas entre 18 y 49. Se podría considerar que son instancias de pequeño tamaño.
  - **Subconjunto Small-1:** formado por 42 de las 84 instancias del conjunto «Small». Este subconjunto se emplea en el contexto de los algoritmos exactos.
  - **Subconjunto Small-2:** formado por todas las instancias del conjunto «Small». Este subconjunto se emplea en el contexto de los algoritmos heurísticos.
- **Grid:** este conjunto está formado por matrices construidas como el producto cartesiano de dos rutas [149]. Son también conocidos como rejillas bidimensionales. La solución óptima al CMP para este tipo de grafos es conocida en función de las dimensiones de la instancia. Para este conjunto de instancias, los vértices del grafo están dispuestos en forma de rejilla rectangular con una dimensión *ancho* x *alto*.
  - **Subconjunto Grid-1:** formado por 36 instancias, en el que el tamaño de las mismas viene definido por $ancho, alto \in \{3, 4, 5, 6, 7, 8, 9, 10\}$, siendo $ancho \geq alto$. Este subconjunto se emplea en el contexto de los algoritmos exactos.
  - **Subconjunto Grid-2:** formado por 81 instancias, en el que el tamaño de las mismas viene definido por $ancho, alto \in \{3, 6, 9, 12, 15, 18, 21, 24, 27\}$. Este subconjunto se emplea en el contexto de los algoritmos heurísticos.
- **Harwell-Boeing**: es un subconjunto de instancias de la colección de matrices dispersas «Harwell-Boeing» (HB) anteriormente mencionada. Esta colección está formada por matrices simétricas provenientes de diferentes disciplinas científicas. Los grafos se derivan de estas matrices, y están formados por tantos vértices como filas tenga la matriz, y un arista (i, j) por cada elemento $M_{ij} \neq 0$.
  - **Subconjunto Harwell-Boeing-1:** formado por 34 instancias del conjunto original, seleccionando aquellas instancias que representan a grafos menores de 200 vértices. El número de aristas oscila entre 46 y 2145. Este subconjunto se emplea en el contexto de los algoritmos exactos.

[1] Disponible en `http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/`

Subconjunto	Número de instancias	Vértices	Aristas	Contexto
Small-1	42	16-24	18-49	Exactos
Grid-1	36	9-100	12-180	Exactos
Harwell-Boeing-1	34	30-199	46-2145	Exactos
Small-2	84	16-24	18-49	Heurísticos
Grid-2	81	9-729	12-1404	Heurísticos
Harwell-Boeing-2	87	30-700	46-41686	Heurísticos

Tabla 5.1: Subconjuntos de instancias empleadas en la experimentación de los algoritmos exactos y heurísticos.

- **Subconjunto Harwell-Boeing-2:** formado por 87 instancias del conjunto original, seleccionando aquellas instancias que representan a grafos menores de 700 vértices. El número de aristas oscila entre 46 y 41686. Este subconjunto se emplea en el contexto de los algoritmos heurísticos.

La principal motivación para la existencia de dos subconjuntos para cada tipo de instancia se debe al tamaño de las mismas. Los algoritmos exactos requieren tiempos de cómputo mucho mayores cuando el tamaño de la instancia es grande, por lo que es común que los conjuntos de datos empleados para la experimentación con estos algoritmos, estén formados por instancias de menor tamaño que aquéllas que forman parte de los conjuntos empleados en la evaluación de los algoritmos heurísticos. Además, cuando un algoritmo de Ramificación y Acotación no es capaz de resolver de manera óptima instancias de un tamaño determinado en un tiempo prefijado, es más que previsible que tampoco lo haga para instancias de mayor tamaño.

Un resumen de los subconjuntos de instancias anteriormente descritos se puede ver en la Tabla 5.1, donde se muestra el número de instancias que contiene cada subconjunto, el tamaño mínimo y máximo (tanto en número de vértices como en número de aristas) de las instancias del subconjunto y el contexto en el que ha sido empleado. Como se puede comprobar en dicha tabla, se ha empleado un total de 112 instancias para la evaluación de los algoritmos exactos. Éstas están divididas en tres conjuntos de entre 34 y 42 instancias cada uno con un número de vértices que oscila entre 9 y 199. Por otro lado, se han empleado un total de 252 instancias para la evaluación de los algoritmos heurísticos, divididas en tres conjuntos con un tamaño de entre 81 y 87 instancias cada uno. El número de vértices de las instancias de estos conjuntos oscila entre 9 y 700. La relación exhaustiva de instancias que componen cada subconjunto se puede consultar en el Apéndice C.

5.1.2. Medidas de calidad

Una vez definidos los conjuntos de instancias sobre los cuales llevar a cabo la experimentación, es necesario determinar una serie de medidas de calidad que permitan discriminar entre unos algoritmos y otros. Si bien es cierto que existen medidas genéricas que pueden ser aplicadas a

muchas disciplinas, en general, éstas dependen del área en el que se esté trabajando. Las medidas de calidad utilizadas en esta Tesis Doctoral y que son empleadas a lo largo de este capítulo, son ampliamente utilizadas en el área de optimización. En particular se han considerado las siguientes medidas: *gap*, desviación respecto al mejor, número de mejores/número de óptimos y tiempo de CPU (del inglés, *Central Processing Unit*). A continuación se describe cada una de ellas de manera detallada.

Comparación con una cota: *gap*

El *gap* determina la diferencia entre una cota superior (UB) y una cota inferior (LB) para una instancia determinada de un problema ($gap = \mid UB - LB \mid$). Es una métrica comúnmente utilizada para medir la calidad de las soluciones obtenidas por los algoritmos exactos.

Dado que el CMP se trata de un problema de minimización, UB se corresponde con el valor de la función objetivo de la mejor solución encontrada, mientras que el LB es una estimación del mínimo valor que puede tomar dicha función objetivo para esa instancia. Si la diferencia entre UB y LB es igual a cero, el valor de la solución encontrada es el óptimo.

El *gap* se expresa habitualmente como un valor normalizado porcentual ($\%gap$). En el caso de que el problema abordado sea un problema de minimización, el $\%gap$ para cada instancia se calcularía mediante la siguiente expresión:

$$\%gap = \frac{UB - LB}{LB} \times 100$$

Una vez calculado el $\%gap$ para cada instancia, es común reportar la media de los $\%gap$ de todas las instancias de un conjunto. Cuanto menor sea el $\%gap$, más ajustadas son las cotas propuestas.

En general, el valor del *gap* sin normalizar es denominado «*gap* absoluto», mientras que al valor normalizado se le denomina «*gap* relativo».

Comparación con la mejor solución: desviación respecto al mejor

La desviación (dev) del valor de la función objetivo de una solución (fo) respecto al mejor valor conocido ($best$) para una instancia, permite determinar lo próxima que está una solución dada, de la mejor solución conocida hasta el momento para esa instancia o, en su defecto, de la solución óptima si ésta fuera conocida. Para un problema de minimización se calcula mediante la siguiente expresión:

$$\%dev = \frac{fo - best}{best} \times 100$$

Al igual que ocurre con el $\%gap$, es común calcular este valor porcentual para cada instancia

de un conjunto por separado y, posteriormente, reportar el valor promediado de todas las instancias del mismo.

Número de óptimos o número de mejores soluciones encontradas

Otra medida que permite comparar distintos algoritmos es el número de soluciones óptimas obtenidas por cada algoritmo comparado (denotado como $\#\acute{o}pt$). Este número se emplea habitualmente en el contexto de los algoritmos exactos, ya que éstos son capaces de determinar si se ha encontrado la solución óptima. Por otro lado, cuando los algoritmos que se desea comparar son algoritmos heurísticos es común emplear el número de mejores soluciones encontradas ($\#mejor$). En general, el número de mejores soluciones se emplea cuando no se conoce la solución óptima, bien porque no se dispone de algoritmos exactos para la resolución del problema o bien porque éstos no han sido capaces de encontrarla para alguna instancia determinada. No obstante, siempre es posible emplear el número de mejores soluciones encontradas en un experimento, con independencia de que los óptimos sean o no conocidos. Tanto el número de óptimos como el número de mejores soluciones encontradas se reportan en valor absoluto.

Tiempo de CPU

La última medida empleada en esta Tesis Doctoral es el tiempo de CPU ($CPUt$) que especifica el tiempo de proceso que necesita un algoritmo determinado para obtener una solución. Es común calcular el tiempo empleado para cada instancia de un conjunto por separado y, posteriormente, reportar el promedio de los tiempos de CPU para todas las instancias del mismo. Si dos algoritmos tienen una calidad promedio igual (medida por ejemplo mediante su desviación al mejor valor conocido) se considera mejor aquél que emplea un menor tiempo en alcanzarla.

Es importante destacar que, cuando se mide el tiempo de CPU de un algoritmo, no se considera el tiempo necesario para cargar las instancias en memoria o el necesario para mostrar los resultados; es decir, únicamente se computa el tiempo empleado por el algoritmo para alcanzar la solución.

5.1.3. Herramientas empleadas

Como se ha indicado anteriormente, un factor que tiene impacto en los resultados obtenidos durante el proceso de experimentación es el ordenador en el que se ha llevado a cabo. Por ello, a la hora de comparar distintos algoritmos es deseable que todos los resultados hayan sido obtenidos empleando el mismo ordenador. En el caso de que esto no fuera posible, se puede establecer una comparación entre las tecnologías empleadas en cada caso, aplicando un factor de corrección a los tiempos. En concreto, todos los experimentos descritos en este capítulo han sido realizados empleando un ordenador Intel Core 2 Quad CPU Q8300 2,5 Ghz, con 6 GB de RAM.

Los algoritmos propuestos en esta Tesis Doctoral han sido implementados empleando el lenguaje de programación Java, concretamente Java SE 6. Además, para el desarrollo de los algoritmos heurísticos, se ha empleado el *framework* «Optsicom Framework»[2]. En el Apéndice A se lleva a cabo una descripción detallada del mismo, así como del «Optsicom Optimization Suite» en el que se integra.

Para la evaluación de la calidad de los resultados obtenidos por los algoritmos exactos, éstos han sido comparados con aquéllos obtenidos por el *solver* CPLEX[3] utilizando la formulación entera descrita en la Sección 2.3.

5.2. Resultados experimentales de los algoritmos exactos

En esta sección se presentan los resultados experimentales obtenidos por los algoritmos exactos propuestos en el Capítulo 3, sobre los conjuntos de instancias «Small-1», «Grid-1» y «Harwell-Boeing-1».

Inicialmente se llevan a cabo una serie de experimentos preliminares (ver Sección 5.2.1) sobre un pequeño porcentaje de las instancias de los conjuntos anteriormente mencionados. Estos experimentos ilustran aspectos tales como la exploración del espacio de soluciones, la aportación de las distintas cotas propuestas a la solución obtenida o la incidencia de la utilización de un algoritmo heurístico para la obtención de una cota superior inicial. Una vez acabada la experimentación preliminar, se realiza la experimentación final (ver Sección 5.2.2) sobre los conjuntos de instancias enteros. Aquí se comparan los resultados obtenidos con aquéllos alcanzados por el *solver* CPLEX empleando la formulación propuesta en [114].

5.2.1. Experimentación preliminar

La experimentación preliminar se ha llevado a cabo sobre cinco instancias representativas de cada uno de los subconjuntos formados para evaluar los algoritmos exactos. En concreto se trata de las siguientes instancias: «p51_20_28», «p63_21_42», «p72_22_49», «p81_23_46» y «p100_24_34» (del conjunto «Small-1») «Grid5x5», «Grid6x8», «Grid7x9», «Grid8x9» y «Grid10x10» (del conjunto «Grid-1») y «ash85», «ibm32», «arc130», «west0167» y «will199» (del conjunto «Harwell-Boeing-1»). A este conjunto, formado por 15 instancias representativas de diferente número de vértices y densidad, se le denomina $TestSet$-1.

El tiempo máximo de ejecución establecido para los experimentos preliminares ($CPUt$) es de 1800 segundos para cada instancia y algoritmo. Cuando un algoritmo no finalice la exploración en el tiempo establecido, se reportan tanto la mejor solución como la mejor cota inferior encontrada.

[2]Puede encontrarse más información en `http://www.optsicom.es/suite/framework`
[3]`http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`

Exploración del árbol de búsqueda

El primer experimento preliminar realizado compara el rendimiento de los tres algoritmos de Ramificación y Acotación propuestos (BB1, BB2 y BB3) sobre el conjunto $TestSet$-1. Para cada uno de ellos, y sobre cada una de las instancias de dicho conjunto, se reporta el número de nodos del árbol que han sido explorados ($Expl.$), el número de nodos del árbol que han sido podados empleando alguna de las cotas propuestas ($Pod.$) y por último el número de nodos que no han sido ni podados ni explorados ($NoExpl.$).

Cuando se ha explorado el árbol al completo, $Pod. + Expl.$ es igual al número total de nodos en el árbol de exploración y, por lo tanto, $NoExpl.$ es igual a cero. En esta situación se puede certificar que el algoritmo ha encontrado la solución óptima.

Los resultados de este experimento se muestran agrupados por tipo de instancia en la Tabla 5.2. A su vez, dentro de cada tipo, las instancias están ordenadas descendentemente por número de vértices. Cuando se observan los resultados instancia por instancia, es posible extraer conclusiones del comportamiento de los algoritmos según las características de cada instancia. Por ejemplo, en función del número de vértices, de la densidad de aristas que posee, etc. En cambio, cuando se reportan valores promedio de un conjunto de instancias, este comportamiento no se aprecia.

		BB1			BB2			BB3		
		Expl.	*Pod.*	*NoExpl.*	*Expl.*	*Pod.*	*NoExpl.*	*Expl.*	*Pod.*	*NoExpl.*
Small (5)	p51_20_28	1,4E04	6,6E18	0,0E0	1,4E04	6,6E18	0,0E0	1,4E04	6,6E18	0,0E0
	p63_21_42	1,2E06	1,4E20	0,0E0	1,2E06	1,4E20	0,0E0	1,2E06	1,4E20	0,0E0
	p72_22_49	1,3E06	3,1E21	0,0E0	1,3E06	3,1E21	0,0E0	1,3E06	3,1E21	0,0E0
	p81_23_46	7,7E06	7,0E22	0,0E0	7,7E06	7,0E22	0,0E0	7,7E06	7,0E22	0,0E0
	p100_24_34	1,5E06	1,7E24	0,0E0	1,5E06	1,7E24	0,0E0	1,5E06	1,7E24	0,0E0
Grid (5)	Grid5x5	6,5E02	4,2E25	0,0E0	6,5E02	4,2E25	0,0E0	6,5E02	4,2E25	0,0E0
	Grid6x8	1,3E04	3,4E61	0,0E0	1,3E04	3,4E61	0,0E0	1,3E04	3,4E61	0,0E0
	Grid7x9	5,9E05	5,4E87	0,0E0	5,9E05	5,4E87	0,0E0	5,9E05	5,4E87	0,0E0
	Grid8x9	3,6E07	3,7E103	1,3E104	3,5E07	2,1E103	1,4E104	3,2E07	1,4E104	3,1E103
	Grid10x10	2,0E07	1,8E148	2,5E158	2,0E07	5,4E142	2,5E158	8,0E05	3,7E157	2,2E158
HB (5)	ibm32	1,6E08	2,9E34	6,9E35	1,5E08	7,1E34	6,4E35	2,5E06	1,3E35	5,9E35
	ash85	4,4E07	4,8E125	7,7E128	4,1E07	1,3E123	7,7E128	4,2E05	2,5E127	7,4E128
	arc130	1,1E07	3,1E197	1,8E220	1,2E07	5,7E148	1,8E220	1,8E05	0,0E0	1,8E220
	west0167	6,0E06	2,6E285	4,1E300	6,6E06	4,7E256	4,1E300	1,1E05	0,0E0	4,1E300
	will199	4,6E06	3,2E318	1,1E373	5,0E06	4,7E256	1,1E373	8,0E04	0,0E0	1,0E373

Tabla 5.2: Nodos explorados, podados y sin explorar en el árbol de búsqueda.

Como se puede observar en la Tabla 5.2, tanto BB1, como BB2 y BB3 son capaces de resolver las cinco instancias del conjunto «Small» de manera óptima. En este caso, tal y como se muestra en las cinco filas correspondientes a estas instancias, el número de nodos explorados y el número de nodos podados es exactamente el mismo para los tres métodos. Dado que la única diferencia entre BB1, BB2 y BB3 es el orden de expansión, este comportamiento se produce cuando el árbol es explorado por completo y no se ha actualizado el valor del UB inicial. No obstante, podría darse el caso en el que, para alguna instancia, alguno de los algoritmos podara más nodos que el resto, pese a que todos ellos resolvieran la instancia de manera óptima. Esto puede ser debido a que un algoritmo sea capaz de alcanzar un nodo hoja con una solución mejor que la solución de partida antes que el resto, actualizando así el valor de su UB con anterioridad, lo que le permitiría podar más ramas. En este caso, el número de nodos podados sería mayor y el número de nodos explorados menor.

En el conjunto de instancias «Grid», se puede observar cómo los tres algoritmos de Ramificación y Acotación son capaces de resolver las instancias con un menor número de vértices «Grid5x5», «Grid6x8» y «Grid7x9». En cambio, en las instancias de mayor tamaño, «Grid8x9» y «Grid10x10», pese a que ninguna de las variantes es capaz de acabar, se puede apreciar un comportamiento diferente entre BB3 y los otros dos algoritmos. Tal y como se muestra en la Tabla 5.2, el número de nodos podados por BB3 sobre estas dos instancias es mucho mayor que el podado por BB1 y BB2. Por lo tanto, se podría afirmar que BB3 ha sido capaz de explorar una mayor parte del árbol.

El último conjunto de instancias considerado es el conjunto «HB». En él, ninguna de las instancias ha podido ser resuelta de manera óptima. Nuevamente, el comportamiento de BB3 frente a BB1 y BB2 se repite en las instancias de menor tamaño, «ibm32» y «ash85», ambas con menos de 100 vértices, para las que BB3 poda un mayor número de nodos. En cambio, llama la atención que, para instancias de tamaño aún mayor, BB3 no sea capaz de podar ningún nodo en el tiempo límite establecido de 1800 segundos, tal y como se muestra en la Tabla 5.2. El motivo principal para este comportamiento en las instancias más grandes es la estrategia de expansión empleada por BB3. Dicha estrategia está basada en la prioridad de los nodos candidato, es decir, en lo prometedores que son. En general, los nodos situados en niveles superiores son más prometedores, ya que la solución parcial asociada a ese nodo tiene menos vértices con una etiqueta asignada, lo que conduce a una expansión similar a una exploración en amplitud del árbol. Esto, a su vez, implica que es necesario expandir muchos nodos en niveles superiores (en instancias grandes la cantidad de nodos en dichos niveles es mucho mayor) antes de alcanzar nodos situados en un nivel en el que las cotas propuestas permitan podar. Todo esto conduce a que los algoritmos BB1 y BB2, cuyo recorrido del árbol de exploración se asemeja a un recorrido en profundidad, empiecen a podar más rápidamente, dado que los recorridos en profundidad permiten alcanzar nodos cuya solución parcial es de peor calidad, que la solución parcial de los niveles superiores.

		BB1			BB2			BB3		
		LB	*gap*	%*gap*	*LB*	*gap*	%*gap*	*LB*	*gap*	%*gap*
Small (5)	p51_20_28	6	0	0,0	6	0	0,0	6	0	0,0
	p63_21_42	12	0	0,0	12	0	0,0	12	0	0,0
	p72_22_49	14	0	0,0	14	0	0,0	14	0	0,0
	p81_23_46	13	0	0,0	13	0	0,0	13	0	0,0
	p100_24_34	7	0	0,0	7	0	0,0	7	0	0,0
Grid (5)	Grid5x5	6	0	0,0	6	0	0,0	6	0	0,0
	Grid6x8	7	0	0,0	7	0	0,0	7	0	0,0
	Grid7x9	8	0	0,0	8	0	0,0	8	0	0,0
	Grid8x9	3	6	200,0	3	6	200,0	8	1	12,5
	Grid10x10	3	8	266,7	3	8	266,7	8	3	37,5
HB (5)	ibm32	6	17	283,3	6	17	283,3	18	6	33,3
	ash85	5	11	220,0	5	11	220,0	9	7	77,8
	arc130	62	140	225,8	62	140	225,8	62	140	225,8
	west0167	10	47	470,0	10	47	470,0	12	45	375,0
	will199	8	134	1675,0	8	134	1675,0	15	123	820,0
Promedio		11,3	24,2	222,7	11,3	24,2	222,7	13,7	21,7	105,5

Tabla 5.3: Cota inferior (*LB*), *gap* absoluto (*gap*) y *gap* relativo (%*gap*) para cada una de las instancias del conjunto *TestSet*-1.

El comportamiento descrito por BB3 en el experimento anterior permite deducir que una temprana finalización del método conduciría a un bajo número de nodos podados. No obstante, esto no significa que el algoritmo no haya sido capaz de visitar una gran cantidad de nodos durante su ejecución, lo que contribuye a mejorar la cota inferior para esa instancia y, por consiguiente, a mejorar también el %*gap* final obtenido por el algoritmo. Para ilustrar este hecho se presenta la Tabla 5.3, en la que se muestra, para cada instancia del conjunto *TestSet*-1, la cota inferior (*LB*), el *gap* y el %*gap*, obtenido por cada método.

Como se puede comprobar en la Tabla 5.3, el *gap* y %*gap* de las instancias para las que se ha encontrado la solución óptima es igual a 0. Además, para estas instancias el *LB* es el mismo, y representa el valor de la solución óptima. En el caso de las instancias para las que los algoritmos no han sido capaces de obtener el óptimo («Grid8x9», «Grid10x10» y todas las del conjunto «HB») el *LB* difiere. En concreto, se aprecian diferencias destacables entre los resultados obtenidos por el algoritmo BB3 y los obtenidos por los algoritmos BB1 y BB2. Se puede comprobar cómo los resultados obtenidos por estos dos últimos algoritmos son idénticos para las 15 instancias consideradas. Esto es debido, fundamentalmente, a que llevan a cabo un recorrido del árbol de exploración muy parecido. En cambio, para BB3 se observa que las *LB* obtenidas son siempre mayores o iguales que las *LB* obtenidas por BB1 y BB2 y, por lo tanto, más ajustadas. Este resultado redunda en que tanto el *gap*, como el %*gap* sea menor. Observando las tres instancias en las que BB3 no había sido capaz de realizar podas en el tiempo de 1800 segundos, tal y como se mostró en la Tabla 5.2, se puede comprobar como, pese a no haber tenido tiempo suficiente para empezar a podar, sí se consigue un mejor *gap* y %*gap* en dos de ellas y un *gap* igual al obtenido por BB1 y BB2 en la tercera.

A raíz de los resultados obtenidos tras estos dos experimentos iniciales, mostrados tanto en la Tabla 5.2 como en la Tabla 5.3, y sintetizando parte de las conclusiones anteriormente extraídas, se puede concluir que el algoritmo BB3 es la mejor estrategia de exploración. Por un lado, es capaz de resolver de manera óptima el mismo número de instancias que las variantes BB1 y BB2 y, por otro, en el caso de las instancias para las que no es capaz de obtener la solución óptima, obtiene un menor *gap* en casi todas las instancias, lo que conduce a su vez a un *gap* promedio más reducido. Por simplicidad, y pese a que la experimentación final se lleva a cabo empleando los tres algoritmos propuestos, se ha seleccionado la variante BB3 para ilustrar los siguientes experimentos previos.

Aportación de las cotas inferiores

En este experimento previo se lleva a cabo un estudio de la aportación de cada una de las cotas propuestas, tanto a la poda de ramas del árbol de exploración durante el proceso de búsqueda, como a la reducción del *gap* final obtenido por el algoritmo cuando no es capaz de finalizar su exploración en el tiempo prefijado.

Inicialmente, se ha contabilizado el porcentaje de nodos que cada una de las cotas es capaz de podar a medida que se va recorriendo el árbol de exploración. Es importante destacar que, dado un nodo del árbol de exploración, existe la posibilidad de que éste pueda ser podado por diferentes cotas. En la Tabla 5.4 se muestra el porcentaje de nodos que es capaz de podar cada una de las cotas propuestas (LB_1, LB_2, LB_3, LB_4 y LB_5) en las 15 instancias del conjunto *TestSet*-1. Los resultados se muestran separados por conjuntos de instancias. Como se ha mencionado anteriormente, se ha empleado la estrategia de exploración BB3 para llevar a cabo este experimento, calculando todas las cotas en cada nodo explorado y, en el caso de que alguna de ellas sea capaz de podar, contabilizando los nodos del árbol que podaría, en función del nivel del mismo en el que se encuentre el nodo podado.

TestSet-1	$\boldsymbol{LB_1}$	$\boldsymbol{LB_2}$	$\boldsymbol{LB_3}$	$\boldsymbol{LB_4}$	$\boldsymbol{LB_5}$
Small (5)	0	93,85	0	96,30	0
Grid (5)	0	99,49	0	98,82	0
HB (5)	0	92,41	0	98,42	0
Promedio	0	95,12	0	98,53	0

Tabla 5.4: Promedio de nodos podados por cada cota inferior.

Los resultados mostrados en la Tabla 5.4 confirman que las cotas LB_1, LB_3 y LB_5 no son capaces de podar un número significativo de nodos cuando se usan en combinación con las cotas LB_2 y LB_4 (los porcentajes asociados son muy próximos a cero y han sido representados con un 0 en la Tabla 5.4 por simplicidad). Por otro lado, se puede observar cómo el comportamiento de LB_2 y LB_4 es muy similar, siendo capaces de podar aproximadamente el 95 % y el 98 %

respectivamente, del total de nodos podados durante el proceso de exploración. Esto lleva a la conclusión de que no es necesario calcular LB_1, LB_3 y LB_5 en cada nodo del árbol de exploración.

No obstante, cuando se alcanza el tiempo límite preestablecido para la exploración del árbol y el algoritmo no es capaz de examinarlo entero, el cómputo de estas cotas conduce a incrementar notablemente la cota inferior para muchas de las instancias, reduciendo así de manera significativa el *gap*. Para demostrar este hecho, se ha llevado a cabo un nuevo experimento en el que se compara el *gap* que obtendría el algoritmo BB3 sin incluir LB_1, LB_3 y LB_5, y el que obtendría incluyendo todas las cotas. Los resultados de este experimento se muestran en la Tabla 5.5, donde BB3 + LB_{Set1} representa a la variante de BB3 que incluye sólo las cotas LB_2 y LB_4 y BB3 + LB_{Set2} representa a la variante de BB3 que incluye todas las cotas. Se proporciona tanto el *gap* absoluto como el relativo para cada subconjunto de instancias del conjunto $TestSet$-1 y el promedio de todo el conjunto.

$TestSet$-1	**BB3 + LB_{Set1}**		**BB3 + LB_{Set2}**	
	gap	%gap	gap	%gap
Small (5)	0,0	0,0	0,0	0,0
Grid (5)	0,8	10,0	0,8	10,0
HB (5)	73,2	447,4	64,8	310,6
Promedio	24,7	152,5	21,9	106,9

Tabla 5.5: Promedio del valor del *gap* para diferentes variantes de BB3 sobre el conjunto de instancias $TestSet$-1.

Los resultados de la Tabla 5.5 muestran que las cotas LB_1, LB_3 y LB_5 ayudan a reducir el *gap* final obtenido por el método. En concreto, el *gap* relativo promedio sobre las instancias del conjunto $TestSet$-1 se reduce del 152,5 % al 106,9 % cuando se incluyen estas cotas. Es importante destacar también, que este efecto habría sido mucho más significativo si cabe, en el caso de que el recorrido del árbol de exploración se hubiera realizado con BB1 o BB2. Esto es debido a que la cota inferior estimada cuando se emplean estas variantes es sustancialmente inferior, debido al tipo de recorrido que llevan a cabo estos algoritmos, tal y como se puede comprobar en la Tabla 5.3.

Examinando los dos experimentos anteriores de manera conjunta, se puede concluir que las cotas inferiores propuestas se complementan unas a otras. Por un lado LB_2 y LB_4 ayudan a podar un gran número de nodos en el árbol de exploración, lo que aumenta notablemente las posibilidades de encontrar la solución óptima en un tiempo determinado, mientras que las cotas LB_1, LB_3 y LB_5 reducen sustancialmente el *gap* final en las instancias en las que el algoritmo no es capaz de encontrar la solución óptima en ese tiempo. Esto justifica la inclusión de todas las cotas propuestas en las distintas variantes del algoritmo de Ramificación y Acotación.

Aportación de la cota superior inicial

El último experimento preliminar se centra en la aportación del método heurístico tipo GRASP (descrito en el Capítulo 3) empleado en el cálculo de una cota superior inicial. Para ello, se ha comparado el rendimiento del algoritmo de Ramificación y Acotación BB3 cuando éste emplea una cota superior inicial calculada con la heurística anteriormente mencionada (se ha identificado esta propuesta en la Tabla 5.6 como BB3+GRASP) con los resultados obtenidos por el mismo BB3 partiendo de una cota superior obtenida a partir del valor de la función objetivo de una solución aleatoria (representado en la Tabla 5.6 como BB3+Aleatoria).

TestSet-1	**BB3 + GRASP**		**BB3 + Aleatoria**	
	gap	%gap	gap	%gap
Small (5)	0,0	0,0	1,4	11,7
Grid (5)	0,8	10,0	33,0	471,4
HB (5)	64,8	310,6	179,4	1055,2
Promedio	21,9	106,9	71,3	512,8

Tabla 5.6: Comparación del impacto de la calidad de la cota superior inicial en el proceso de exploración, sobre el conjunto de instancias *TestSet*-1.

Los resultados de la comparativa anteriormente descrita, y que son expuestos en la Tabla 5.6, avalan el uso de un algoritmo heurístico para el cálculo de una cota superior inicial. Los resultados de BB3 junto con la cota superior inicial calculada con la heurística GRASP son sustancialmente mejores que los obtenidos partiendo de una solución aleatoria. De manera específica, los primeros obtuvieron un *gap* absoluto promedio de 21,9 y un *gap* relativo de 106,9 %, mientras que los obtenidos por la versión que partía de una solución aleatoria, obtuvieron un *gap* absoluto de 71,3 y un *gap* relativo de 512,8 %. Observando los resultados por conjuntos de instancias, en todos ellos el comportamiento se repite, siendo mejor la variante BB3+GRASP.

Además del *gap* relativo y absoluto, se ha computado también el número de instancias del conjunto *TestSet*-1 para las cuales la heurística tipo GRASP fue capaz de obtener la solución óptima. Este hecho no se puede certificar para todas las instancias del conjunto, ya que no para todas ellas se conoce el valor de la solución óptima. En concreto, el óptimo es conocido para las 5 instancias tipo «Small» (ya que éstas han sido resueltas de manera óptima por las distintas variantes del algoritmo de Ramificación y Acotación) y para las 5 instancias tipo «Grid» (ya que el óptimo para este tipo de instancias se conoce por diseño [159]). Cabe destacar que para las 10 instancias del conjunto *TestSet*-1 para las que el óptimo es conocido, el heurístico fue capaz de alcanzarlo en todas ellas. No obstante, en cuanto a las 5 instancias del conjunto «HB» se refiere, no es posible determinar si la solución encontrada por el heurístico es óptima.

5.2.2. Experimentación final

Una vez llevada a cabo la experimentación preliminar descrita en la Sección 5.2.1 en la que se ha ilustrado el comportamiento de los algoritmos de Ramificación y Acotación propuestos, en aspectos tales como: la exploración del espacio de soluciones; la aportación de las distintas cotas a la solución obtenida o el impacto del uso de un algoritmo heurístico para la obtención de una cota superior inicial, se lleva a cabo la experimentación final sobre los conjuntos de instancias completos «Small-1», «Grid-1» y «Harwell-Boeing-1».

Comparación con CPLEX

Cada una de las tres variantes del algoritmo de Ramificación y Acotación propuesto son comparadas, no sólo entre sí, sino también con los resultados obtenidos por el *solver* CPLEX (versión 11.1) empleando la formulación descrita en el Capítulo 3 y que fue introducida en [114]. El modelado necesario para representar dicha formulación de modo que CPLEX pueda interpretarla se ha llevado a cabo empleando la API para Java del propio *solver*.

Las tres variantes descritas (BB1, BB2 y BB3) incluyen las cinco cotas inferiores propuestas (LB_1, LB_2, LB_3, LB_4 y LB_5) y la cota superior inicial empleada en el proceso de exploración ha sido calculada empleando la heurística tipo GRASP. Es importante destacar también que el valor de la cota superior, obtenido por el heurístico, ha sido facilitado a CPLEX al comienzo del proceso de búsqueda.

En la Tabla 5.7 se muestra el número de soluciones óptimas encontradas por cada uno de los algoritmos (#Ópt.), el promedio del *gap* absoluto (*gap*), el promedio del *gap* relativo (%*gap*) y el promedio del tiempo de CPU en segundos (CPUt) para cada conjunto de instancias. En total se han empleado 112 instancias para este experimento: 42 pertenecientes al conjunto «Small-1», 36 pertenecientes al conjunto «Grid-1» y 34 pertenecientes al conjunto «Harwell-Boeing-1». Al igual que en los experimentos previos, se ha limitado el tiempo de CPU máximo para cada algoritmo e instancia a 1800 segundos.

Analizando los resultados expuestos en la Tabla 5.7 se observa que, para el conjunto «Small-1», el *solver* CPLEX (empleando la formulación lineal entera para el problema [114]) únicamente es capaz de obtener el óptimo en 9 instancias de las 42 que forman el conjunto en el tiempo límite de 1800 segundos. En concreto, se trata de aquellas instancias de menores o iguales a 20 vértices. Por otro lado, para el mismo conjunto de instancias, las tres variantes del algoritmo de Ramificación y Acotación propuestas, claramente mejoran los resultados alcanzados por CPLEX empleando esta formulación, siendo capaces de obtener el óptimo para todas ellas en tiempos de cómputo que, en promedio, se sitúan entre 2 y 5 segundos.

En cuanto al conjunto de instancias «Grid-1» se refiere, el *solver* CPLEX fue capaz de encontrar el valor óptimo para 2 de las 36 instancias, mientras que las tres variantes de los

		BB1	BB2	BB3	CPLEX
«Small-1» (42)	#Ópt.	42	42	42	9
	gap	0,0	0,0	0,0	1,9
	%*gap*	0,0	0,0	0,0	54,7
	CPUt	2,1	2,2	4,9	1573,9
«Grid-1» (36)	#Ópt.	30	30	30	2
	gap	1,1	1,0	0,28	4,2
	%*gap*	37,1	29,5	3,5	211,1
	CPUt	301,5	301,5	302,4	1707,9
«HB-1» (34)	#Ópt.	5	5	4	0
	gap	51,8	51,8	49,7	97
	%*gap*	314,4	314,4	210,3	634,8
	CPUt	1574,7	1573,9	1594,4	1800,0

Tabla 5.7: Comparación de los algoritmos de Ramificación y Acotación propuestos con los resultados obtenidos por CPLEX empleando la formulación descrita en la Sección 2.3.

algoritmos de Ramificación y Acotación propuestas fueron capaces de obtener el óptimo en 30 de las 36 instancias en un tiempo promedio de unos 300 segundos. Además, observando el promedio del *gap* relativo, se observa como BB3 tiene un %*gap* promedio de un 3,5 %. Este valor es inferior al obtenido por los algoritmos BB1 y BB2, con un promedio del 37,1 % y del 29,5 % respectivamente, y muy inferior al obtenido por CPLEX con un 211,1 %.

Por último, en el conjunto «Harwell-Boeing-1», compuesto por instancias de mayor tamaño, el *solver* CPLEX no fue capaz de obtener la solución óptima para ninguna de las instancias del conjunto, mientras que los algoritmos propuestos BB1, BB2 y BB3 fueron capaces de obtener 5, 5 y 4 óptimos respectivamente. En referencia al %*gap* promedio, nuevamente BB3 obtuvo un *gap* relativo promedio inferior a BB1 y BB2 (210,3 % frente a 314,4 %) y muy inferior al alcanzado por CPLEX, con un 634,8 %.

A tenor de los resultados mostrados en la Tabla 5.7 se podría concluir que los algoritmos de Ramificación y Acotación propuestos (BB1, BB2 y BB3) tienen un comportamiento similar sobre los conjuntos de instancias evaluados, obteniendo prácticamente el mismo número de soluciones óptimas. En referencia al *gap* alcanzado por los algoritmos, cuando éstos no son capaces de acabar su ejecución en un tiempo prefijado, se puede afirmar que BB3 tiene un rendimiento ligeramente superior a BB1 y BB2. Pese a ello, cabe destacar que los tres algoritmos propuestos mejoran los resultados obtenidos por el *solver* CPLEX empleando la formulación lineal entera reportada en [114].

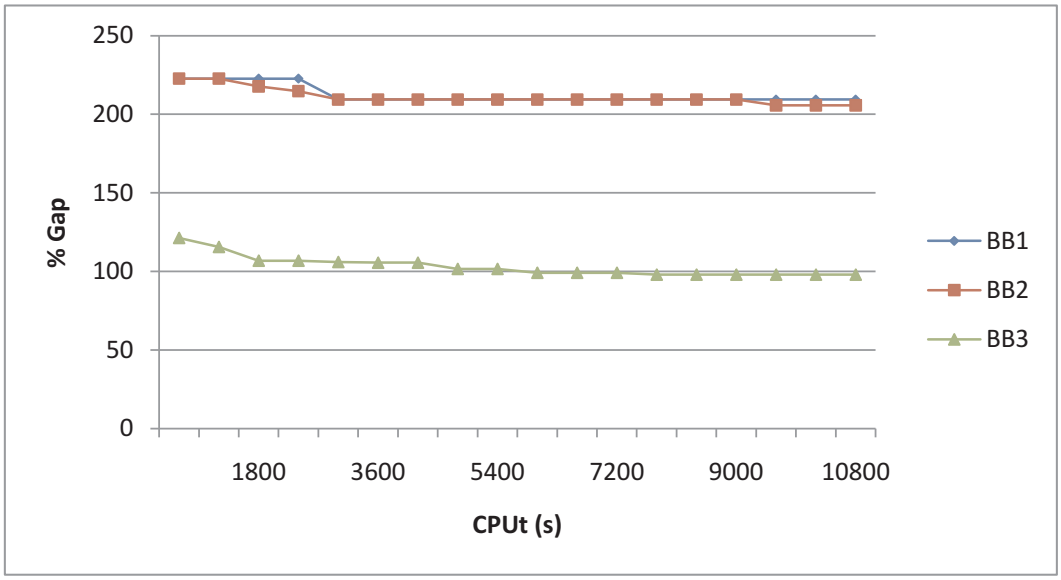


Figura 5.1: Perfil de búsqueda de los algoritmos BB1, BB2 y BB3 sobre las instancias del conjunto «*TestSet*-1».

Perfil de búsqueda

En el último experimento realizado en relación con los algoritmos de Ramificación y Acotación propuestos, se presenta el perfil de búsqueda de las tres variantes BB1, BB2 y BB3, al ser ejecutadas durante 3 horas sobre las 15 instancias del conjunto «*TestSet*-1». El resultado de esta comparativa se muestra en la Figura 5.1, donde se ha representado la progresión del *gap* relativo promedio con muestras cada 10 minutos. Los puntos extraídos se han unido con una línea, observando así la tendencia.

La progresión mostrada en la Figura 5.1 confirma que BB3 obtiene mejores resultados que BB1 y BB2, en cuanto al *gap* se refiere. Por otro lado, se puede observar cómo la reducción del *gap* más significativa se produce durante los primeros 30 minutos de ejecución, obteniendo una mejora marginal en el tiempo restante.

5.2.3. Mejores cotas conocidas para el conjunto «Harwell-Boeing-1»

Observados los resultados obtenidos sobre los distintos conjuntos de instancias considerados, se puede concluir que las 34 instancias del conjunto «Harwell-Boeing-1» podrían considerarse como las más difíciles abordadas en la evaluación de los algoritmos de Ramificación y Acotación propuestos. Por ello, en el Apéndice D se muestran las mejores cotas superiores e inferiores encontradas para las instancias de este conjunto. Estos valores son el resultado de ejecutar el heurístico tipo GRASP durante 10 minutos para la obtención de una cota superior inicial y de ejecutar los algoritmos de Ramificación y Acotación durante 4 horas para la obtención de la cota inferior. Los valores obtenidos pueden ser empleados como marco de referencia en futuras

comparaciones.

5.3. Resultados experimentales de los algoritmos heurísticos

En esta sección se presentan los resultados experimentales obtenidos por los algoritmos heurísticos propuestos en el Capítulo 4, sobre los conjuntos de instancias «Small-2», «Grid-2» y «Harwell-Boeing-2» descritos en la Sección 5.1.1.

Inicialmente se lleva a cabo una serie de experimentos preliminares (ver Sección 5.3.1) sobre un pequeño porcentaje de las instancias de algunos de los conjuntos anteriormente mencionados, que ilustran aspectos tales como el rendimiento de los algoritmos constructivos, el impacto del ajuste de parámetros en la búsqueda local, las diferencias entre los métodos de combinación propuestos, la influencia de la composición y tamaño del *RefSet*, etc.

Una vez acabada la experimentación preliminar, se realiza la experimentación final (ver Sección 5.3.2) sobre los conjuntos de instancias enteros, comparando los resultados obtenidos, con aquéllos alcanzados por los algoritmos heurísticos encontrados en el estado del arte, y que se encuentran descritos en la Sección 2.4.

5.3.1. Experimentación preliminar

La experimentación preliminar se ha llevado a cabo sobre 10 instancias representativas del conjunto «Grid-2» y 10 instancias representativas del conjunto «Harwell-Boeing-2». En concreto se trata de las siguientes instancias: «Grid6x21», «Grid9x21», «Grid9x24», «Grid9x27», «Grid12x24», «Grid12x27», «Grid21x9», «Grid24x12», «Grid27x9» y «Grid27x12» (del conjunto «Grid-2») y «bcsstm07», «dwt__361», «dwt__592», «fs_680_1», «lund_a», «lund_b», «pores_3», «saylr3», «steam1» y «steam2» (del conjunto «Harwell-Boeing-2»). A este conjunto, formado por 20 instancias de diferente número de vértices y densidad, se le denomina $TestSet$-2.

Además de ilustrar algunos de los aspectos (anteriormente mencionados) de los métodos propuestos, uno de los objetivos principales de los experimentos preliminares es el ajuste de parámetros de los algoritmos. Para ello, es importante que las instancias empleadas en dichos experimentos preliminares sean representativas. En esta ocasión no se han seleccionado instancias del conjunto «Small-2» para realizar los experimentos previos, debido a que son las de menor tamaño y, por lo tanto, es más difícil que permitan discriminar entre los algoritmos.

Comparativa de algoritmos constructivos

El primer experimento previo realizado se centra en la comparación de los diferentes algoritmos constructivos propuestos en la Sección 4.3, denominados C1, C2, C3 y C4. El objetivo principal de esta comparativa es determinar cuál, de los algoritmos anteriormente mencionados,

formará parte del esquema de Búsqueda Dispersa (SS, del inglés *Scatter Search*) que se pretende construir. Este experimento debe permitir, por lo tanto, aportar información relevante que ayude en la selección del algoritmo, por lo que es importante determinar las características deseables de las soluciones construidas. Está bien documentado (ver por ejemplo [100]) que las soluciones iniciales del esquema de SS deben ser soluciones de buena calidad, pero también suficientemente diversas entre sí, de modo que permitan a la búsqueda local y a los métodos de combinación alcanzar diferentes óptimos locales.

Para comparar los algoritmos constructivos se han realizado 100 construcciones con cada uno de los métodos propuestos, sobre cada instancia del conjunto «$TestSet$-2». Empleando esas 100 construcciones por instancia se ha calculado un único valor que represente la calidad de las soluciones aportadas por ese constructivo y un único valor que represente su diversidad.

Para obtener el valor que representa la calidad de las soluciones de un constructivo, inicialmente se ha reportado, para cada instancia, el promedio del valor de la función objetivo de las 100 soluciones creadas con ese constructivo, resultando un total de 20 promedios (uno por cada instancia). Estos 20 valores han sido nuevamente promediados entre las 20 instancias del conjunto «$TestSet$-2», obteniendo así un único valor de calidad para cada constructivo sobre las instancias de ese conjunto.

Para calcular un valor que represente la diversidad de las soluciones de un constructivo, inicialmente se han considerado, para cada instancia por separado, las soluciones creadas por ese constructivo. Para ello, se han formado conjuntos con las 100 soluciones creadas por instancia y constructivo. De cada conjunto, se han ido tomando las soluciones una a una y se ha calculado la distancia de ésta al conjunto, empleado la medida de distancia introducida en la Sección 4.5.1 y considerando la distancia de una solución a un conjunto como la menor de las distancias entre la solución evaluada y alguna de las soluciones del conjunto. Este valor se ha promediado para las 100 soluciones, obteniendo así un valor de distancia por instancia y constructivo. Por último, se han promediado todos los valores obtenidos para cada constructivo sobre las instancias del conjunto «$TestSet$-2».

En la Figura 5.2 se muestra una gráfica en la que se representa el valor de la calidad de las soluciones obtenidas por cada constructivo, para las instancias del conjunto $TestSet$-2 (Promedio Calidad) frente a la diversidad de las mismas (Promedio Diversidad). Se aprecia cómo el constructivo con mejor calidad (nótese que dado que se trata de un problema de minimización, cuanto menor sea la media del valor de la función objetivo, mejor es la calidad del algoritmo) es el constructivo C2, mientras que el que obtiene un peor promedio de calidad es el constructivo C4. En cambio, al prestar atención a la diversidad de las soluciones, se observa justo lo contrario, el constructivo C4 es el que proporciona soluciones más diversas, mientras que C2 es el que proporciona soluciones menos diversas. En general, se observa que cuanto más diversas son las soluciones encontradas por un algoritmo constructivo, menor calidad promedio tienen y

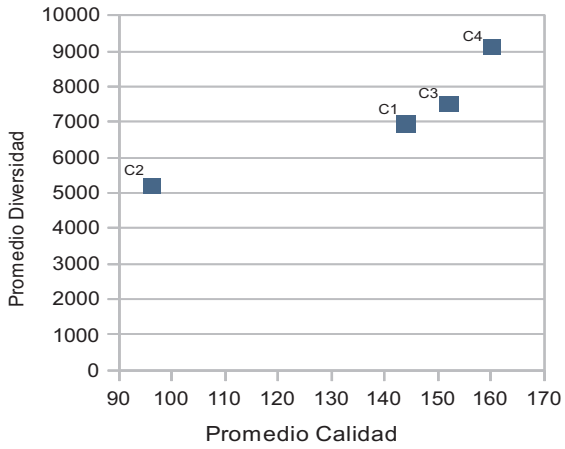


Figura 5.2: Promedio de la calidad frente a la diversidad de las soluciones construidas por cada algoritmo sobre las instancias del conjunto «$TestSet$-2».

viceversa.

En función de los resultados anteriores, es difícil decidir qué constructivo seleccionar para formar parte del esquema de SS, debido a que no es posible premiar calidad y diversidad al mismo tiempo, siendo deseable encontrar un balance entre ambas. No obstante, en el esquema de SS, las soluciones construidas son posteriormente mejoradas con una búsqueda local. Esto puede hacer que soluciones de menor calidad se conviertan en soluciones mejores y que los resultados de diversidad obtenidos varíen. Por ello, la decisión de la selección del constructivo se ha relegado al siguiente experimento previo, en el que se evalúa el impacto de la búsqueda local sobre las soluciones construidas por cada algoritmo.

Soluciones obtenidas tras el proceso de construcción y mejora

Una vez examinados los algoritmos constructivos de manera independiente, se pasa a observar la combinación de éstos con el método de búsqueda local propuesto en la Sección 4.4. Es importante considerar la combinación de constructivo y búsqueda local ya que, en ocasiones, la influencia de la búsqueda local en las soluciones obtenidas por distintos constructivos, puede hacer que un algoritmo constructivo que inicialmente parece mejor, en realidad no lo sea. Esto puede suceder, por ejemplo, si el constructivo es demasiado voraz o si las soluciones se parecen mucho entre sí (algo poco deseable en un esquema SS). En ocasiones, sucede que al aplicar una búsqueda local a las soluciones construidas, éstas convergen rápidamente a óptimos locales de baja calidad de los que es difícil escapar. Cabe la posibilidad, por lo tanto, de que un constructivo que obtenga soluciones con una calidad promedio inferior a otro, sea mejor al combinar ambos con un procedimiento de mejora.

Para evaluar este aspecto, se ha emparejado cada uno de los algoritmos constructivos considerados (C1, C2, C3 y C4) con el método de búsqueda local propuesto en la Sección 4.4.

	Promedio Calidad	Mejor	Promedio Diversidad	CPUt
C1 + Búsqueda Local	80,1	62,6	3746,4	287,34
C2 + Búsqueda Local	84,3	61,8	3993,5	295,25
C3 + Búsqueda Local	112,9	81,3	5555,8	435,41
C4 + Búsqueda Local	114,1	79,0	5811,3	388,41

Tabla 5.8: Comparación del impacto de la búsqueda local sobre las soluciones producidas por cada constructivo en el conjunto de instancias $TestSet$-2.

Dicha búsqueda local está configurada de manera exhaustiva, es decir, todos los vértices son candidatos a ser movidos y todas las posiciones son candidatas a ser destino de un vértice que se mueve. Nuevamente se han realizado 100 construcciones para cada instancia y constructivo y éstas han sido mejoradas empleando el procedimiento de búsqueda local. Al igual que en el experimento previo anterior (que contemplaba únicamente los algoritmos constructivos) en la Tabla 5.8 se ha reportado el promedio de la calidad y el promedio de la diversidad de las soluciones obtenidas tal y como se describió anteriormente. Además, se muestra también el promedio del mejor valor encontrado para cada instancia, por cada combinación de constructivo y búsqueda local y el tiempo invertido en construir y mejorar 100 soluciones.

Los resultados aportados en la Tabla 5.8 muestran que los algoritmos C1 y C2 (emparejados con el procedimiento de búsqueda local) obtienen resultados notablemente mejores que los alcanzados por los algoritmos C3 y C4, tanto en el promedio de las 100 construcciones (Promedio Calidad) como en el promedio de la mejor solución encontrada (Mejor). Por otro lado, C3 y C4 son mejores respecto a la diversidad de las soluciones encontradas, aunque el consumo de tiempo es sustancialmente mayor en ambas variantes. La justificación para este comportamiento reside en que, dado que la calidad de las soluciones inicialmente construidas es peor, la búsqueda local emplea más tiempo mejorando. Los argumentos de calidad y tiempo hacen que se descarten los constructivos C3 y C4.

Prestando atención a la combinación de los constructivos C1 y C2 con la búsqueda local, se puede observar que, tanto los valores promedio de la calidad, como el tiempo de CPU, son muy parecidos tanto en C1 como en C2. Sin embargo, las soluciones obtenidas empleando C2 tienen una mayor diversidad. Además, observando el promedio del valor de la función objetivo de la mejor solución encontrada, también es mejor. Por ello, se ha decidido seleccionar el algoritmo constructivo C2 para ser incluido en el esquema de SS definitivo.

Ajuste de parámetros de la búsqueda local

La configuración de la búsqueda local en el experimento anterior implicaba que todos los vértices de la solución eran candidatos a ser insertados en otra posición y todas las posiciones eran susceptibles a ser destino de un vértice insertado. Sin embargo, tanto el número de vértices a desplazar, como el número de posiciones a tener en cuenta en cada inserción, pueden considerarse

parámetros del procedimiento de búsqueda. De hecho, es interesante prestar atención a estos parámetros, ya que es posible que se estén realizando muchos movimientos que no supongan una mejora de la solución de la que se dispone o que ésta sea marginal y, para alcanzarla, se esté empleando una gran cantidad de tiempo de CPU.

En este experimento previo se lleva a cabo una evaluación de los parámetros β y ω. El parámetro β determina el número de elementos en la lista de candidatos a ser movidos (denominada lista de vértices críticos). Está expresado como un porcentaje sobre el *cutwidth* máximo de la solución considerada. Un valor de β pequeño implica que el tamaño de la lista de vértices críticos será mayor que con un valor de β grande. Por ejemplo, si β fuera igual a 0,2, implicaría que entran en la lista de vértices críticos todos aquellos vértices que tengan un *cutwidth* por encima del 20 % del valor del *cutwidth* máximo de esa solución.

El parámetro ω, en cambio, determina el máximo número de posiciones que se considerarán como posibles destinos de un vértice crítico. Está expresado como un porcentaje sobre el número de vértices del grafo. Un valor de ω pequeño implica considerar menos posiciones, como posiciones destino de un vértice crítico, que un valor de ω grande. Por ejemplo, si ω fuera igual a 0,2, el 20 % de las posiciones del tamaño del grafo serían seleccionadas como candidatas a ser destino de un vértice crítico, siguiendo los criterios descritos en la Sección 4.4, .

En concreto, se han considerado tres valores distintos para β (0,8, 0,5 y 0,2) y otros tres valores para ω (0,1, 0,3 y 0,5). Dentro de los valores considerados para cada parámetro, la combinación de un valor de β pequeño, con un valor de ω grande, indica que el grado de exploración es lo mayor posible. Análogamente, un valor de β grande y un valor de ω pequeño indica que el grado de exploración es lo menor posible.

En la Tabla 5.9 se presentan los resultados obtenidos empleando diferentes configuraciones de la búsqueda local (descrita en la Sección 4.4) sobre las instancias del conjunto $TestSet$-2. Cada par de datos reportado (Promedio, CPUt) se corresponde con una configuración distinta de los parámetros β y ω de la búsqueda local. Para obtener dichos valores se han realizado 100 construcciones (empleando el constructivo seleccionado, C2) y 100 mejoras, reportando el promedio del valor de la función objetivo de la mejor solución encontrada para cada instancia (Promedio) y el tiempo de CPU empleado (CPUt) también promediado.

Observando fila a fila la Tabla 5.9, se puede comprobar cómo a medida que disminuye el valor de β aumenta el tiempo de CPU, ya que se realizan inserciones con un mayor número de vértices críticos. Por otro lado, observando los datos por columnas, se aprecia cómo, a medida que se aumenta el valor de ω, aumenta también el CPUt, ya que los vértices críticos considerados se prueban en más posiciones candidatas.

En general, cuanto mayor es la exploración llevada a cabo (más vértices críticos considerados y más posiciones candidatas evaluadas) se obtiene un mejor promedio del valor de la función

ω \ β		0,8	0,5	0,2
0,1	Promedio	89,08	86,53	85,82
	CPUt	28,65	58,67	66,06
0,3	Promedio	88,88	86,01	85,31
	CPUt	82,24	161,17	179,67
0,5	Promedio	88,88	85,98	85,28
	CPUt	115,52	229,14	230,31

Tabla 5.9: Estudio del impacto de los parámetros de la búsqueda local β y ω, en el promedio del valor de la función objetivo y en el tiempo de CPU (en segundos).

objetivo, aunque el tiempo empleado también es mayor. No obstante, el aumento del tiempo de CPU no siempre va asociado a una mejora del valor de la función objetivo. Por ejemplo, si se comparan los resultados obtenidos con $\beta = 0{,}8$ y $\omega = 0{,}3$ y los obtenidos con $\beta = 0{,}8$ y $\omega = 0{,}5$ se puede comprobar cómo, pese a que el tiempo de CPU ha aumentado considerablemente (pasando de 82,24 a 115,52 segundos) el valor promedio de la función objetivo no ha mejorado (manteniéndose en 88,88). Éste es un ejemplo claro en el que aumentar la exploración (y en consecuencia el tiempo de ejecución) no ha supuesto una mejora en la función objetivo.

En ocasiones, la modificación de los parámetros β y ω permite ahorrar una cantidad considerable de tiempo de ejecución, con una degradación moderada del valor de la función objetivo. Se pretende, por lo tanto, seleccionar una combinación de valores de los parámetros β y ω capaz de, en un tiempo de CPU razonable, obtener un Promedio lo más bajo posible. Observando los resultados obtenidos con el aumento del valor del parámetro ω, se aprecia como éste tiene un impacto mayor en el incremento del tiempo de CPU que en la mejora del valor de la función objetivo. Por ello se ha decidido tomar un valor $\omega = 0{,}1$. Una vez fijado el valor de ω, se pasa a observar la combinación de éste con los distintos valores de β. En concreto, se aprecia una ganancia considerable entre $\beta = 0{,}8$ (con un Promedio igual a 89,08) y $\beta = 0{,}5$ (con un Promedio igual a 86,53) mientras que esta ganancia no es tan grande cuando se compara $\beta = 0{,}5$ con $\beta = 0{,}2$ (con un Promedio igual a 85,82).

Siguiendo los razonamientos anteriores se ha tomado la decisión de configurar la búsqueda local empleando un valor para $\beta = 0{,}5$ (se considerarán vértices críticos aquéllos cuyo *cutwidth* esté por encima del 50 % del valor de la función objetivo para esa solución) y un valor $\omega = 0{,}1$ (se probará la inserción de los vértices críticos en un máximo de posiciones del 10 % del número de vértices del grafo). Estos parámetros de configuración han sido empleados en los siguientes experimentos previos.

Métodos de combinación

En este experimento previo se comparan los tres métodos de combinación propuestos en la Sección 4.5.2. Para llevar a cabo el experimento se ha configurado un algoritmo de SS con

cada uno de los métodos propuestos, y se ha ejecutado durante 20 iteraciones (considerando una iteración como el conjunto de pasos necesarios llevados a cabo por el algoritmo hasta que se produce una actualización o reconstrucción del *RefSet*) sobre las instancias del conjunto $TestSet$-2. Cada algoritmo SS está formado por el constructivo C2 como método de generación de soluciones diversas, la búsqueda local descrita en la Sección 4.4 configurada tal y como se ha descrito en el experimento anterior ($\beta = 0,5$ y $\omega = 0,1$) como método de mejora y el método de combinación de soluciones correspondiente (CM1, CM2 o CM3).

En la Tabla 5.10 se muestra el promedio de la desviación respecto al mejor valor conocido para cada instancia (Desv.), el número de mejores soluciones encontradas (#Mejor) y el tiempo de CPU en segundos (CPUt) para cada variante de SS configurada con cada uno de los métodos de combinación propuestos (CM1, CM2 y CM3) y denotada como SS+CM1, SS+CM2 y SS+CM3 respectivamente.

	SS + CM1	**SS + CM2**	**SS + CM3**
Desv.(%)	6,08 %	5,58 %	4,33 %
#Mejor	8	9	10
CPUt(s)	351,63	298,54	241,02

Tabla 5.10: Comparativa de los diferentes métodos de combinación empleados en un esquema SS sobre las instancias del conjunto de instancias $TestSet$-2.

Los resultados de la Tabla 5.10 muestran que, con respecto a la desviación promedio y al número de mejores soluciones, SS+CM3 proporciona los mejores resultados, con una desviación del 4,33 % y un total de 10 mejores soluciones encontradas. Los otros métodos de combinación tienen una desviación mayor y un menor número de mejores soluciones. Además, el tiempo consumido por SS+CM3 es ligeramente inferior al empleado por SS+CM1 y SS+CM2. Por lo tanto, CM3 será el método de combinación empleado en el esquema de SS definitivo.

Mejora selectiva de las soluciones

Típicamente, en el esquema de SS, cada solución obtenida por el método de generación de soluciones diversas, o por el método de combinación, es sometida al procedimiento de mejora. Considerando que la ejecución de este procedimiento es computacionalmente costosa (supone el procedimiento que más tiempo necesita, de los incluidos en el esquema de SS) emplearlo sobre cada solución generada puede resultar demasiado lento. Por ello, se ha examinado el impacto en la calidad de las soluciones, al aplicar la búsqueda local únicamente a un subconjunto de las soluciones creadas por el método de generación de soluciones diversas o por el método de combinación de soluciones, realizando así una aplicación selectiva de la búsqueda local. En concreto, en la aplicación selectiva de la búsqueda local, ésta se aplicaría a las b mejores soluciones (construidas o combinadas) donde $b = \mid RefSet \mid$. El objetivo es comprobar si la calidad de las

	Mejora Exhaustiva			Mejora Selectiva		
$\mid b \mid$	**Desv.**	**#Mejor**	**CPUt**	**Desv.**	**#Mejor**	**CPUt**
5	5,34 %	9	100,45	8,07 %	6	12,45
10	4,33 %	10	241,02	7,65 %	6	22,73
15	4,59 %	10	502,33	7,35 %	7	37,01

Tabla 5.11: Comparativa entre la mejora selectiva de soluciones y la mejora exhaustiva de las mismas, empleando el conjunto de instancias *TestSet*-2, para diferentes tamaños de *RefSet*.

soluciones obtenidas al mejorar sólo algunas de ellas se resiente en exceso. En caso contrario, se debe evaluar la ganancia en tiempo, lo que permitiría realizar más iteraciones del algoritmo de SS en un contexto en el que, por ejemplo, el tiempo estuviera prefijado.

En la Tabla 5.11 se muestran los resultados de la mejora selectiva frente a la mejora exhaustiva para diferentes valores de b. Los resultados revelan que, en el contexto del CMP, la mejora selectiva no parece una buena estrategia, teniendo en cuenta que la mejora exhaustiva obtiene mejores resultados de manera sistemática (tanto en desviación, como en número de mejores soluciones) que la mejora selectiva. Como era de esperar, el tiempo de CPU empleado por la mejora selectiva es sustancialmente inferior al empleado por la mejora exhaustiva, pero la diferencia en la calidad de las soluciones obtenidas es demasiado grande.

En cuanto al impacto del tamaño del *RefSet* (considerando la mejora exhaustiva) se observa que, de entre los tres tamaños evaluados (5, 10 y 15 soluciones) se obtienen los mejores resultados empleado el tamaño considerado estándar ($b = 10$).

Diseño factorial para distintos parámetros de búsqueda

En los experimentos previos anteriormente descritos, se han empleado distintos parámetros de búsqueda. Para establecer algunos de ellos, ha sido necesario fijar otros. El objetivo de este experimento es evaluar distintas combinaciones de algunos de esos parámetros, junto con un parámetro nuevo, realizando para ello un experimento factorial. Este experimento debe permitir determinar si existen diferencias significativas en los resultados, al emplear diferentes parámetros clave en el algoritmo SS. Para ello, se ha empleado una ANOVA multifactor con cuatro parámetros y tres niveles para cada parámetro, tal y como se muestra en la Tabla 5.12. Los tres primeros parámetros expresados en dicha tabla (β, b y CM) pueden identificarse en los experimentos anteriores. Para complementarlos, se ha añadido un nuevo parámetro (q) que determina la composición del conjunto de referencia, en cuanto al porcentaje de soluciones añadidas por calidad y al porcentaje de soluciones añadidas por diversidad se refiere. En la literatura relacionada con SS, estos porcentajes se establecen típicamente al 50 %, añadiendo así al *RefSet* la mitad de las soluciones por calidad y la otra mitad por ser diversas respecto a las primeras. En este experimento, en cambio, se han evaluado tres posibles valores para q (10 %, 50 % y 90 %) donde $q = 10\,\%$ significa que $0{,}1 \cdot b$ soluciones son incluidas en el *RefSet*

	β	b	CM	q
Nivel 1	0,2	5	CM1	10 %
Nivel 2	0,5	10	CM2	50 %
Nivel 3	0,8	15	CM3	90 %

Tabla 5.12: Diseño factorial para distintos parámetros de búsqueda.

debido a su calidad y el resto $(0{,}9 \cdot b)$ debido a su diversidad.

Considerando el porcentaje de desviación con respecto a la mejor solución conocida como variable dependiente y β, b, CM y q como factores, se ha aplicado una ANOVA donde el modelo está limitado a los principales efectos y para el que se han considerado todas las interacciones de dos vías. Aplicado este diseño factorial a las 20 instancias del conjunto *TestSet*-2, se obtienen un total de 1620 ejecuciones. Los resultados obtenidos determinaron que β es el parámetro con mayor influencia en el algoritmo, seguido por los métodos de combinación (ambos presentaron un F-valor grande, de 50,7 y 8,6 respectivamente, con un nivel de significación de $0,00 < 0,05$). Por otro lado, la influencia de b y q en los resultados del algoritmo es mucho más limitada, con un nivel de significación mayor de 0,05. La interacción más consistente se da entre β y b con un F-valor de 8,7 y un nivel de significación de 0,00, seguido por la interacción entre β y CM, con un F-valor de 3,4 y un nivel de significación de 0,01.

La mejor configuración de los parámetros considerados en este experimento está compuesta por $\beta = 0,2$, $b = 15$, CM = CM3 y $q = 0,9$, combinación que obtuvo una desviación del 3,92 % respecto a los mejores valores conocidos para las instancias del conjunto *TestSet*-2. No obstante, si se considera un $b = 10$ y se mantienen los valores para los otros parámetros, la desviación promedio sólo se incrementa hasta el 4,01 %, pero el tiempo de CPU asociado a cada ejecución se reduce de 446,5 segundos a 238,9 segundos. Este he hecho conduce a seleccionar esta última configuración con un $b = 10$ para los experimentos finales.

Proceso de búsqueda

El objetivo de este último experimento previo es ilustrar la contribución de los distintos elementos de SS en la evolución de la mejor solución encontrada. En particular, la Figura 5.3 muestra, para cada iteración de SS, el valor de la mejor solución en el *RefSet* (Mejor *RefSet*), el valor de la mejor solución obtenida con el método de combinación de soluciones en el *RefSet* (Mejor Combinación) y el valor de la mejor solución obtenida tras aplicar la búsqueda local a las soluciones combinadas (Mejor Búsqueda L.) para una única instancia.

En la Figura 5.3, se puede apreciar la contribución del método de combinación y del método de mejora. En general, las soluciones obtenidas como combinación de otras no suelen ser mejores que aquéllas que se emplearon para obtenerlas, por lo que el método de combinación por sí sólo, no permite obtener soluciones que mejoren la mejor solución en el *RefSet*. No obstante, se puede

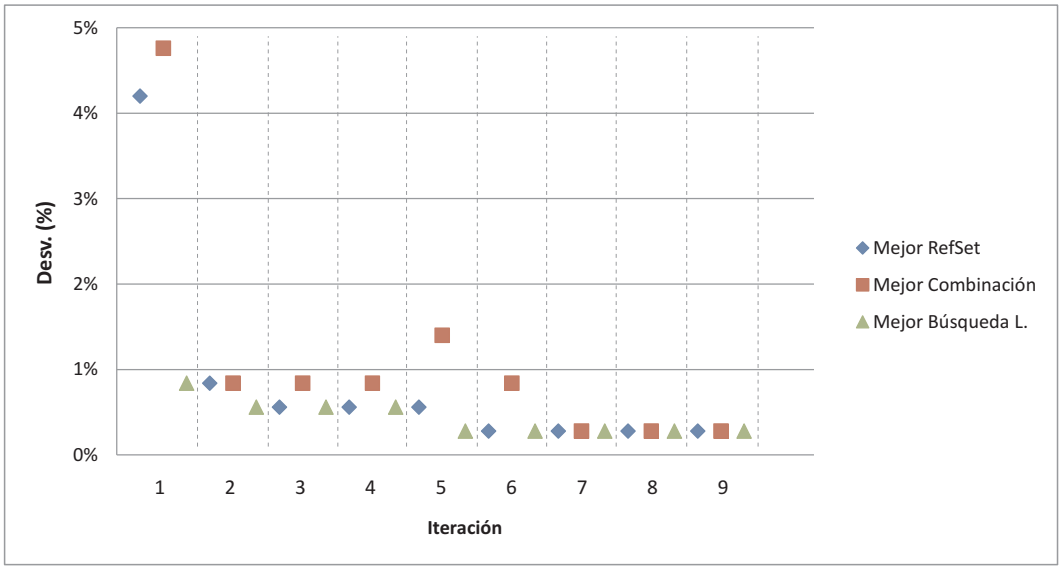


Figura 5.3: Aportación de los distintos elementos de SS a la mejor solución encontrada.

apreciar cómo las soluciones combinadas sí que constituyen buenas soluciones de partida para ser mejoradas por la búsqueda local, algo que se observa especialmente en las iteraciones 1 y 5, donde la aplicación de la búsqueda local a las soluciones combinadas permite mejorar la mejor solución en el *RefSet*, generando así una nueva mejor solución global.

5.3.2. Experimentación final

Los experimentos previos descritos en la Sección 5.3.1 han servido para ilustrar algunos aspectos del método de SS propuesto. Asimismo, han sido utilizados para ajustar los distintos parámetros que componen el método, sobre un pequeño subconjunto de instancias. Una vez finalizado el ajuste de parámetros, el algoritmo debe ser evaluado comparándolo con los métodos del estado del arte (ver Sección 2.4) sobre los conjuntos de instancias «Small-2», «Grid-2» y «Harwell-Boeing-2» descritos en la Sección 5.1.1.

Comparación con los métodos del estado del arte

En esta sección se compara el diseño final de SS con el método basado en *Simulated Annealing* propuesto en [40] representado en las distintas tablas de resultados mediante el acrónimo SA, y con el método basado en GRASP con *Path Relinking* propuesto en [3] y representado mediante el acrónimo GPR.

De acuerdo con los experimentos previos realizados, el método de SS está configurado con $\beta = 0{,}2$, $b = 10$, CM = CM3, $\omega = 0,1$ y $q = 0,9$ y, el número de iteraciones por instancia que se ejecutará el método antes de detenerse está establecido en 20. Debe recordarse que se considera que una iteración ha finalizado tras una actualización o una reconstrucción del *RefSet*.

En los experimentos finales realizados, los métodos del estado del arte han sido configurados con los parámetros recomendados por sus autores y se han ejecutado durante un tiempo similar al empleado por SS. Las tablas 5.13, 5.14 y 5.15 muestran los resultados obtenidos por cada método considerado (SS, SA y GPR) para los tres conjuntos de instancias mencionados. En concreto, se ha reportado el promedio del valor de la función objetivo (Promedio), el porcentaje de la desviación a la mejor solución conocida o al óptimo si éste está disponible (Desv.), el número de mejores soluciones (#Mejor) o nuevamente, si los óptimos se conocen, el número de óptimos (#Opt.) y el tiempo de CPU promedio (CPUt) en segundos empleado por cada algoritmo.

	SS	**SA**	**GPR**
Promedio	4,92	5,15	5,20
Desv.(%)	0,00	5,60	6,54
#Opt.	84	64	60
CPUt	0,07	0,07	0,07

Tabla 5.13: Comparación de los algoritmos SS, SA y GPR sobre el conjunto de instancias «Small-2» (84 instancias).

	SS	**SA**	**GPR**
Promedio	13	16,14	38,44
Desv.(%)	7,76	25,42	201,81
#Opt.	44	37	2
CPUt	210,07	216,13	235,16

Tabla 5.14: Comparación de los algoritmos SS, SA y GPR sobre el conjunto de instancias «Grid-2» (81 instancias).

	SS	**SA**	**GPR**
Promedio	315,22	346,21	364,83
Desv.(%)	1,42	48,97	90,97
#Mejor	59	8	2
CPUt	430,56	435,41	557,48

Tabla 5.15: Comparación de los algoritmos SS, SA y GPR sobre el conjunto de instancias «Harwell-Boeing-2» (87 instancias).

En los resultados presentados en las tablas 5.13, 5.14 y 5.15 se aprecia cómo el algoritmo propuesto (SS) obtiene mejores resultados sobre los tres tipos de instancias consideradas, tanto en promedio de la función objetivo, como en desviación y en número de mejores soluciones/óptimos encontrados.

En concreto, en el conjunto «Small-2» (Tabla 5.13) el algoritmo SS es capaz de obtener los 84 valores óptimos en un tiempo promedio inferior a un segundo, mientras que SA y GPR obtienen

64 y 60 óptimos respectivamente. No obstante, si se permite ejecutar los métodos SA y GPR durante un tiempo mayor, ambos son capaces de obtener los 84 valores óptimos en menos de 5 segundos de media. Es importante destacar que para este conjunto de instancias se conocen los valores óptimos, ya que estas instancias fueron resueltas por los algoritmos exactos propuestos en el Capítulo 3, así como por el *solver* CPLEX empleando la formulación propuesta en [114].

Observando los resultados obtenidos sobre las instancias conjunto «Grid-2» (Tabla 5.14) se puede concluir que éstas son sustancialmente más difíciles de resolver que las instancias del conjunto «Small-2». En particular, el método SS fue capaz de encontrar la solución óptima en 44 de las 81 instancias, en un tiempo medio de 210,07 segundos, mientras que SA y GPR obtuvieron 37 y 2 soluciones óptimas respectivamente en un tiempo de ejecución similar. Para este tipo de instancias el valor de la solución óptima es conocido por definición.

Por último, sobre las instancias del conjunto «Harwell-Boeing-2» (Tabla 5.15) para las que la solución óptima no es conocida para la mayoría de las instancias, el algoritmo SS obtuvo 59 veces la mejor solución conocida, mientras que los algoritmos SA y GPR la obtuvieron 8 y 2 veces respectivamente. En el Apéndice D se muestran los mejores valores conocidos para las instancias del conjunto «Harwell-Boeing-2».

Para complementar la información aportada en las tablas 5.13, 5.14 y 5.15 se ha llevado a cabo el test de Friedman para muestras emparejadas, empleando los datos utilizados en estas tablas. El p-valor resultante de 0,000 de este experimento muestra que existen diferencias estadísticamente significativas entre los tres métodos evaluados. En concreto, se está utilizando un nivel de significación de 0,05 como umbral para decidir si se rechaza o no la hipótesis nula. Un típico análisis posterior consiste en obtener un *ranking* de los métodos bajo comparación, de acuerdo con el promedio de valores computados con este test. Según los resultados obtenidos con este último análisis, el mejor método es SS (con un valor de clasificación de 1,36), seguido por SA (con un valor de clasificación de 2,05) y, por último, GPR (con un valor de clasificación de 2,59).

A continuación se han comparado los algoritmos SS y SA empleando dos test no paramétricos bien conocidos para comparación por pares: el test de Wilcoxon y el test de Signos. El primero permite determinar si las dos muestra consideradas representan a poblaciones diferentes. En este caso, las muestras son las soluciones obtenidas con cada método comparado. El p-valor resultante de 0,000 indica que los valores comparados provienen de métodos diferentes. Por otro lado, el test de Signos se emplea para computar el número de instancias en las que un algoritmo supera al otro. El p-valor resultante de 0,000 indica que SS es claro ganador entre ambos métodos.

Evolución de los métodos en el tiempo

En este último experimento se compara la evolución de los tres métodos (SS, SA y GPR) en el tiempo, sobre las instancias del conjunto *TestSet*-2. Para ello, se ejecutaron los tres métodos durante 250 segundos por instancia, reportando la mejor solución encontrada cada 10 segundos. Los resultados de este experimento se muestran en la Figura 5.4. Además, de manera individualizada se presenta la evolución del método SS propuesto en la Figura 5.5.

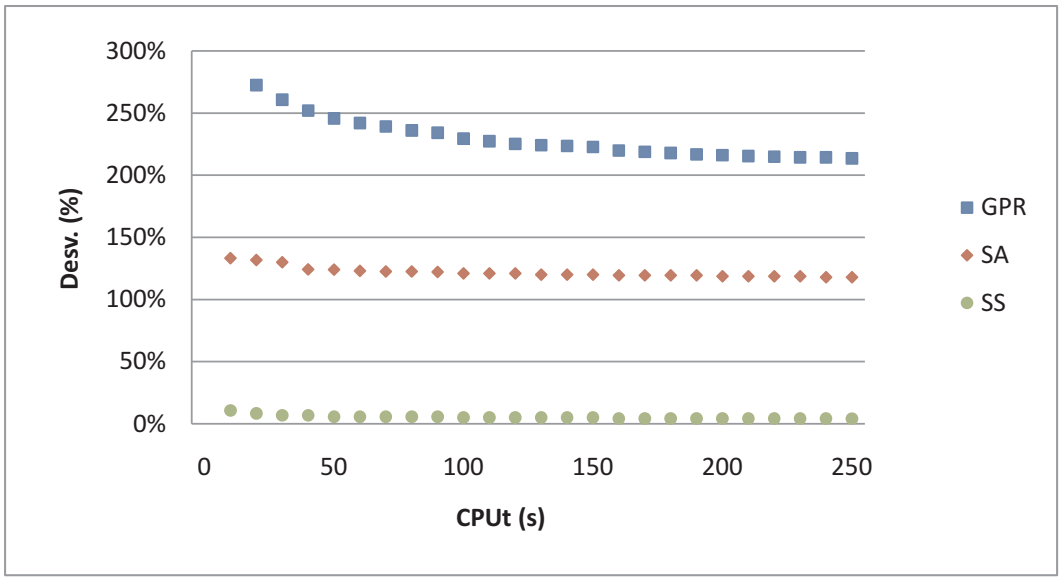


Figura 5.4: Evolución de los métodos SS, SA y GPR a lo largo del tiempo.

En la Figura 5.4 se puede observar cómo SS obtiene soluciones de gran calidad desde el comienzo del proceso de búsqueda. En concreto, en los primeros 10 segundos, presenta una desviación promedio del 10,68 %, mientras que SA y GPR muestran una desviación del 133,31 % y 332,14 % respectivamente. Se puede apreciar como los tres métodos mejoran sus correspondientes desviaciones con el progreso del tiempo, obteniendo una desviación final, a los 250 segundos de ejecución, de 4,08 % (SS), 117,98 % (SA) y 213,64 % (GPR).

Debido a las grandes diferencias existentes entre la desviación del algoritmo SS y las desviaciones de los algoritmos SA y GPR, mostradas en la Figura 5.4, la escala empleada no permite apreciar la mejoría de SS a lo largo de los 250 segundos considerados. En la Figura 5.5 se muestra una nueva gráfica en la que se refleja únicamente la desviación de SS, donde se puede observar como la desviación promedio del método pasa de 10,68 % a 4,08 %. Se aprecia también como la ganancia más sustancial se produce en los primeros 50 segundos, evolucionando más lentamente a partir de ese instante.

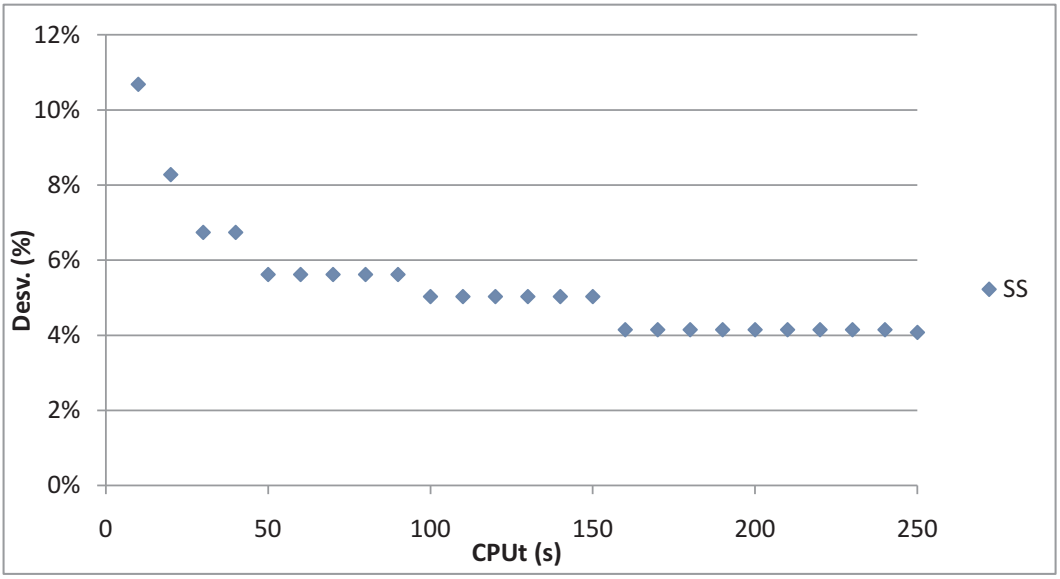


Figura 5.5: Evolución de SS a lo largo del tiempo.

Capítulo 6

Conclusiones y trabajos futuros

En el último capítulo de esta Tesis Doctoral se recogen las conclusiones y principales aportaciones que pueden extraerse de la investigación desarrollada, en relación con el problema de la minimización de la anchura de corte en ordenaciones lineales. Se recogen también los trabajos que han sido publicados durante el proceso de investigación. Por último, se presentan una serie de líneas de trabajo que quedan abiertas y que permitirán continuar con la investigación iniciada en esta Tesis Doctoral.

6.1. Conclusiones

Los problemas NP-Difíciles suponen uno de los grandes retos de la actualidad para el mundo de la computación, tanto desde el punto de vista teórico, como desde el punto de vista tecnológico. La resolución óptima de estos problemas es una barrera insalvable cuando el tamaño del problema es grande, lo que hace justicia al nombre que se emplea frecuentemente para denominarlos: «problemas intratables».

La aparición de más y más problemas con interés práctico, que podrían ser englobados dentro de la categoría NP-Difícil, ha conducido al desarrollo de técnicas eficientes que permiten abordarlos. Algunas de ellas, permiten encontrar soluciones óptimas cuando el tamaño de las instancias de entrada no es demasiado grande (algoritmos exactos). No obstante, para la mayoría de los problemas de la vida real, de mayor tamaño, no es posible encontrar la solución óptima en un tiempo de cómputo razonable, empleando algoritmos exactos. Por ello, existe la necesidad de disponer de otro tipo de técnicas que, pese a que no sean capaces de certificar si las soluciones encontradas son óptimas, obtengan soluciones de alta calidad, en tiempos de cómputo reducidos (algoritmos heurísticos).

En esta Tesis Doctoral se han propuesto tanto algoritmos exactos como heurísticos para la resolución del problema de la minimización de la anchura de corte en ordenaciones lineales. En la Sección 6.1.1 se detallan las principales aportaciones de esta Tesis Doctoral.

6.1.1. Principales aportaciones

En el desarrollo de esta Tesis Doctoral se ha abordado el problema de la minimización de la anchura de corte en ordenaciones lineales. Para ello se ha realizado un completo estudio del estado del arte del problema y se han propuesto tanto algoritmos exactos, como algoritmos heurísticos para su resolución. Estos algoritmos suponen algunas de las principales aportaciones de esta Tesis Doctoral. En particular, en cuanto a la resolución exacta del problema se refiere, se han propuesto diferentes estrategias basadas en la técnica de Ramificación y Acotación, acompañadas de una serie de cotas inferiores y un algoritmo heurístico para el cálculo de una cota superior inicial. Por otro lado, para la obtención de soluciones aproximadas al mismo, han sido propuestas diferentes heurísticas constructivas, de búsqueda local y de combinación de soluciones, englobadas dentro de un esquema metaheurístico de Búsqueda Dispersa. A continuación se detallan, separadas por capítulos, éstas y otras aportaciones destacadas:

- En el Capítulo 1 se lleva a cabo una introducción al problema de la minimización de la anchura de corte en ordenaciones lineales. En él, se describe el problema y sus aplicaciones prácticas, el contexto en el que está enmarcada esta Tesis Doctoral, su motivación, hipótesis de trabajo y objetivos. Además, se esboza la propuesta algorítmica que se desarrolla en capítulos posteriores.

- En el Capítulo 2 se ha realizado un exhaustivo estado del arte del problema abordado, prestando especial atención a los trabajos relacionados directamente con la propuesta de esta Tesis Doctoral: algoritmos exactos y algoritmos heurísticos para la resolución del problema de la minimización de la anchura de corte en ordenaciones lineales. En concreto, se ha llevado a cabo un estudio minucioso de una formulación entera para el problema [114] y de dos propuestas heurísticas [40, 3] basadas en la técnicas de *Simulated Annealing* y GRASP con *Path Relinking* respectivamente. Se recogen también las relaciones conocidas entre el problema abordado y otros problemas de optimización.

- En el Capítulo 3 se repasan las principales técnicas exactas disponibles para la resolución de problemas de optimización. A continuación, se proponen tres algoritmos basados en la técnica de Ramificación y Acotación para la resolución exacta del problema de la minimización de la anchura de corte en ordenaciones lineales. Cada una de las tres variantes planteadas se caracteriza por realizar un recorrido del árbol de exploración diferente. Se proponen también cinco cotas inferiores que son embebidas en los algoritmos anteriores. Algunas de ellas sirven para podar nodos del árbol de exploración, ahorrando así una cantidad considerable de tiempo de ejecución, mientras que otras son de utilidad en la reducción del *gap* final obtenido por los algoritmos, para instancias para las que no se puede encontrar la solución óptima en el tiempo establecido. Por último, se ha propuesto un primer algoritmo heurístico tipo GRASP para el cálculo de una cota superior inicial al

problema, que ha ayudado a mejorar el rendimiento de los algoritmos de Ramificación y Acotación.

- En el Capítulo 4 se repasan las principales técnicas existentes para la obtención de soluciones aproximadas a problemas de optimización. De entre las técnicas estudiadas, se han escogido las metaheurísticas y, más concretamente, el esquema de Búsqueda Dispersa, para la creación de un algoritmo capaz de obtener soluciones aproximadas de buena calidad, al problema de la minimización de la anchura de corte en ordenaciones lineales. En concreto, se han propuesto cuatro heurísticas constructivas tipo GRASP, una búsqueda local y tres métodos de combinación de soluciones. Finalmente, se ha configurado un algoritmo de Búsqueda Dispersa empleando la mejor combinación de método constructivo, búsqueda local, método de mejora y método de combinación de soluciones, de entre las distintas propuestas. Para ello se han tenido en cuenta criterios de calidad y diversidad de las soluciones obtenidas, así como de tiempo de ejecución, al emplear distintas combinaciones de los algoritmos anteriormente descritos.

- En el Capítulo 5 se lleva a cabo una extensa experimentación que ha permitido validar tanto los algoritmos exactos propuestos en el Capítulo 3 como los algoritmos heurísticos propuestos en el Capítulo 4. Para la evaluación de los algoritmos se han empleado tres conjuntos de instancias diferentes (ver Apéndice C). Tanto en la experimentación relacionada con los algoritmos exactos, como la relacionada con los algoritmos heurísticos, se ha seleccionado un pequeño porcentaje de instancias de los conjuntos anteriormente mencionados para llevar a cabo diversos experimentos preliminares. Estos experimentos han ilustrado la aportación de los distintos componentes introducidos en los algoritmos y han servido para ajustar algunos de los parámetros de los mismos. Por último, las versiones finales de los algoritmos propuestos han sido comparadas favorablemente con otras propuestas del estado del arte, avalando así la calidad de los mismos.

6.1.2. Publicaciones

En esta sección se recogen las publicaciones obtenidas a partir del trabajo desarrollado en esta Tesis Doctoral. Algunos de los resultados parciales que se han ido alcanzando durante el desarrollo de este trabajo de investigación han sido publicados en congresos nacionales e internacionales del área:

- Publicaciones en congresos internacionales:

  - «*The Cutwidth Minimization Problem*». Juan J. Pantrigo, Abraham Duarte, Rafael Martí and Eduardo G. Pardo. ALIO/INFORMS Joint International Meeting. Buenos Aires, 2010.

- Publicaciones en congresos nacionales:
  - «*Búsqueda Dispersa para el problema de Ordenación Lineal de Corte Mínimo*». Eduardo G. Pardo, Abraham Duarte y Juan J. Pantrigo. VII Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB). Páginas 85-92. Valencia, 2010. ISBN: 978-84-92812-58-5.
  - «*Optsicom Optimization Suite, un conjunto de herramientas para la investigación en optimización*». Micael Gallego, Francisco Gortázar y Eduardo G. Pardo. VII Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB). Páginas 329-336. Valencia, 2010. ISBN: 978-84-92812-58-5.

Además de los trabajos ya publicados, anteriormente mencionados, los resultados finales presentados en esta Tesis Doctoral y, por lo tanto, principales aportaciones de la misma, han sido sometidos a procesos de revisión por parte de la comunidad científica. En concreto se han enviado dos artículos a revistas indexadas en el *Journal Citation Reports* (JCR), que se encuentran en periodo de evaluación (concretamente en segunda revisión):

- «*Branch and bound for the Cutwidth Minimization Problem*». Rafael Martí, Juan J. Pantrigo, Abraham Duarte and Eduardo G. Pardo. Enviado a *Computers & Operations Research*. Enviado en Junio 2010. Primera revisión en Abril 2011.
- «*Scatter Search for the Cutwidth Minimization Problem*». Juan J. Pantrigo, Rafael Martí, Abraham Duarte and Eduardo G. Pardo. Enviado a *Annals of Operations Research*. Enviado en Julio 2010. Primera revisión en Marzo 2011.

6.2. Trabajos futuros

En el desarrollo de una Tesis Doctoral es frecuente que se identifiquen líneas de trabajo abiertas, que pueden ser una alternativa como punto de partida para futuras investigaciones. A raíz del proceso de investigación iniciado en esta Tesis Doctoral se proponen los siguientes trabajos futuros:

- El problema de la minimización de la anchura de corte en ordenaciones lineales está estrechamente relacionado con otros problemas de optimización (en el Capítulo 2 se recogen algunas de estas relaciones) y, más concretamente, con otros problemas de ordenación lineal. En ocasiones, cuando hay problemas relacionados o alguna forma clara de transformación entre ellos, es posible que las cotas propuestas para un problema sean válidas para otros. En concreto, es interesante estudiar si las cotas propuestas para el CMP son válidas para otros problemas relacionados. De igual manera, tiene interés identificar cotas para otros problemas y comprobar si son válidas para el CMP.

- Como anteriormente se ha mencionado, abordar la resolución óptima de problemas NP-Difíciles es una tarea muy costosa desde el punto de vista computacional, cuando el tamaño del problema es grande. En esta Tesis Doctoral se han empleado instancias de hasta 200 vértices para evaluar los algoritmos de Ramificación y Acotación propuestos. Para las instancias de mayor tamaño no se ha alcanzado su resolución óptima al utilizar una CPU convencional, en los horizontes de tiempo de cómputo considerados. Sería interesante estudiar la posibilidad de implementar los algoritmos de Ramificación y Acotación propuestos en arquitecturas gráficas de procesamiento vectorial y evaluar qué tamaño de problema podría llegar a resolverse para el CMP empleando estos algoritmos. Este aspecto puede tener un gran interés en aplicaciones reales.

- Los algoritmos de Ramificación y Acotación propuestos han sido comparados con la formulación para el CMP propuesta en [114] empleando el *solver* CPLEX. Sería de gran interés estudiar la posibilidad de crear una nueva formulación ILP para el problema. Además, la formulación existente podría ser de utilidad para la proposición de nuevas cotas al problema.

- Pese al justificado interés de emplear una metaheurística poblacional para la obtención de soluciones aproximadas al CMP, sería interesante estudiar las ventajas que podría aportar una metaheurística trayectorial para este problema.

- En la estrategia de búsqueda local propuesta (ver Sección 4.4) uno de los aspectos más determinantes en la eficacia del método fue la definición del criterio de aceptación de un movimiento, que no se limita a la mejora del valor de la función objetivo. Esta idea es especialmente interesante cuando se trabaja con problemas en los que existe una gran cantidad de soluciones en las que el valor de la función objetivo no varía, careciendo así de información que permita determinar la conveniencia de ciertos movimientos. Por lo tanto, esta idea puede ser aplicada a otros problemas en los que la función objetivo consista en minimizar un valor máximo («*Min-Max*») o viceversa.

- En la metodología de Búsqueda Dispersa, la actualización del *RefSet* pasa por ser uno de los aspectos clave en el diseño del algoritmo. En particular, es deseable que el *RefSet* esté formado por soluciones de calidad, pero lo más diversas posible. Por ello, es de interés el estudio de nuevas medidas de distancia entre soluciones que permitan medir la diversidad, así como de la definición de nuevos criterios de entrada en el *RefSet*.

- Se han empleado tres métodos de combinación de soluciones basados en el mecanismo de combinación por votos. El primero podría considerarse el mecanismo por votos clásico y los otros dos son variantes derivadas de la evaluación de las soluciones implicadas. Sería interesante estudiar, por un lado, la aplicabilidad de estas estrategias a otros problemas

de ordenación y, por otro lado, el estudio de nuevos mecanismos genéricos de combinación de soluciones, cuando la estructura de las mismas es una ordenación.

- El concepto de «soluciones de influencia» (véase [73]) puede ser de utilidad para los métodos de combinación. Cuando se combinan dos o más soluciones es posible que, sistemáticamente, existan soluciones cuya combinación genere estructuras más atractivas que otras; es decir, a partir de las cuales se puedan alcanzar óptimos locales de mejor calidad. Es interesante, por lo tanto, estudiar qué soluciones produjeron estas estructuras un mayor número de veces, es decir, qué soluciones son más influyentes. Esto puede jugar un papel determinante en la composición del *RefSet*.

Apéndice A

Optsicom Optimization Suite

Optsicom Optimization Suite (`http://www.optsicom.es/suite`) es un conjunto de herramientas, aplicaciones y servidores que proporcionan soporte al proceso de investigación en optimización [60]. Esta *suite* ofrece funcionalidades complementarias al desarrollo de algoritmos, como la ejecución remota de experimentos, el análisis de los datos obtenidos, o la gestión documental del proceso. Los principales módulos que la componen son los siguientes:

- ***Optsicom Framework***: librería destinada al desarrollo de algoritmos exactos y aproximados para problemas de optimización. Incluye no sólo esquemas algorítmicos para el desarrollo de los mismos, sino también un sistema de configuración y ejecución de experimentos.

- ***Optsicom Experiment Analyzer***: aplicación que proporciona la funcionalidad necesaria para realizar uno o más análisis sobre los datos sin procesar, generados con el módulo de experimentación del *framework*. Dispone de formatos predefinidos de tablas, permitiendo agrupar los valores obtenidos para un conjunto de instancias, emplear diversos estadísticos bien conocidos, etc.

- ***Optsicom Remote Experiment System***: sistema para la ejecución remota de experimentos que permite: seleccionar una o varias máquinas para la ejecución en remoto, configurar experimentos, planificar las máquinas que se utilizarán en cada experimento, gestionar el estado de la ejecución, etc.

- ***Optsicom Eclipse Plugin***: *plugin* para Eclipse que convierte este entorno de programación de propósito general en un entorno de desarrollo de algoritmos para problemas de optimización. Incluye la funcionalidad descrita en el módulo *Optsicom Remote Experiment System*.

- ***Optsicom Interactive Application***: aplicación de escritorio para la ejecución de experimentos y realización de análisis de resultados. Permite, de una manera sencilla

y visual, realizar comparativas entre algoritmos sin necesidad de utilizar un entorno de desarrollo.

- ***Optsicom Content Management System*** (CMS): servidor para los problemas desarrollados con Optsicom Optimization Suite que proporciona acceso centralizado a las instancias de los diferentes problemas, sistema de control de versiones para el código desarrollado, una interfaz web para la presentación de los resultados obtenidos, y otras funcionalidades.

En esta Tesis Doctoral se ha empleado *Optsicom Framework* para el desarrollo de los algoritmos heurísticos propuestos en el Capítulo 4. Además, se ha colaborado en la depuración y evolución del mismo. A continuación se detallan algunas de sus principales características.

Optsicom Framework

Optsicom Framework nace en el contexto de la tesis doctoral de Micael Gallego [59] y, su evolución, podría considerarse el germen de Optsicom Optimization Suite, en el que se integra [60]. Es una librería Java que proporciona un entorno donde los desarrolladores pueden incorporar nuevos algoritmos y ejecutar experimentos con éstos u otros ya existentes. El *framework* oculta al desarrollador los aspectos internos relativos a la ejecución de experimentos, permitiéndole centrarse exclusivamente en las estrategias a desarrollar. Está dividido en los siguientes módulos: core, approx, exact, experiment y utils, que pasan a detallarse a continuación:

- El módulo core contiene las definiciones básicas necesarias para un problema dado. Proporciona las clases necesarias para realizar una especificación de las instancias, dar forma a las soluciones y describir el problema en sí.
- El módulo approx contiene los bloques básicos de construcción de algoritmos aproximados. Los algoritmos se construyen mediante el patrón de diseño *Template Method*, heredando de la clase `ApproxMethod`. En la definición de nuevos algoritmos existen tres enfoques:
  - Construir el algoritmo desde cero, definiendo el procedimiento de alto nivel (en el caso de una metaheurística) así como los de bajo nivel (método constructivo, método de mejora, método de combinación, etc.).
  - Reutilizar uno de los procedimientos de alto nivel proporcionados por el *framework* (`ScatterSearch`, `PathRelinking`, `MultiStart`, etc.) y proporcionar únicamente los procedimientos de bajo nivel.
  - Emplear un enfoque mixto, copiando un procedimiento de alto nivel, modificándolo para las necesidades concretas del algoritmo diseñado, y proporcionando los procedimientos correspondientes de bajo nivel.

El módulo approx también contiene las clases necesarias para integrar en el *framework*, *solvers* comerciales como Evolver[1], OptQuest[2], etc. Estos y otros *solvers* permiten resolver, de forma genérica, problemas de optimización.

- El módulo exact es el equivalente al módulo approx para algoritmos exactos. Este módulo dispone de las clases necesarias para definir algoritmos de ramificación, cotas y modelos que pueden ser utilizados en conjunción con herramientas como CPLEX[3], Gurobi[4] o GLPK[5].
- El módulo experiment contiene la lógica necesaria para realizar la experimentación, permitiendo comparar los diferentes algoritmos desarrollados. Para la definición de experimentos, el desarrollador indica los algoritmos a comparar, sus parámetros, las instancias sobre las que ejecutarlos y el número de ejecuciones por instancia. Es posible indicar también un tiempo límite de ejecución. Es importante notar que el *framework* es capaz de guardar la evolución temporal del algoritmo, almacenando la información relativa a cada uno de los instantes en que se mejoró la mejor solución que había encontrado el algoritmo hasta el momento.
- Por último, el módulo utils contiene clases de utilidad para el resto del *framework* y para el desarrollador, como gestión de generadores de números aleatorios, la gestión de colecciones ordenadas, algoritmos de ordenación, etc.

[1] http://www.palisade.com/evolver/
[2] http://www.opttek.com/Products/OptQuest.html
[3] http://www.ilog.com/products/cplex/
[4] http://www.gurobi.com/
[5] http://www.gnu.org/software/glpk/

Apéndice B

Algunos *solver* disponibles

La resolución exacta de un problema de optimización podría verse como una sucesión de pasos en la que inicialmente se identifica y define el problema a resolver, a continuación se modela y por último se resuelve mediante un *software* específico.

Modelar un problema consiste en obtener una representación del mismo (denominada «modelo») que capte sus componentes clave. Un modelo no es más que una abstracción del problema, por lo que es posible modelar un mismo problema de muchas formas diferentes.

Los modelos que representan problemas de optimización pueden ser plasmados empleando para ello la programación matemática. Existen lenguajes algebraicos genéricos de modelado para programación matemática, que permiten expresar un modelo matemático con independencia del *software* que se emplee posteriormente para su resolución. En la Tabla B.1 se muestran algunos de estos lenguajes.

Lenguaje	**URL**
AMPL	`http://www.ampl.com/`
GAMS	`http://www.gams.com`
MLP	`http://www.maximalsoftware.com/`

Tabla B.1: Lenguajes algebraicos de modelado.

Una vez que se dispone de un modelo para un problema, éste pasa a resolverse empleando un *software* adecuado. Para ello es posible emplear algoritmos genéricos previamente implementados en herramientas o desarrollar algoritmos específicos para cada problema concreto.

Existe una gran cantidad *software*, tanto comercial como libre, que implementa métodos exactos bien conocidos para la resolución de modelos de optimización. Estos paquetes *software* o librerías son denominados *solver*. En la Tabla B.2 se recogen algunos de los *solver* más conocidos, junto con su URL (del inglés, *Uniform Resource Locator*) que permiten la resolución exacta de modelos de optimización. La mayoría de estos *solver* soportan modelos creados con los lenguajes

Solver	**URL**
CPLEX	`http://www.ibm.com/software/integration/optimization/cplex-optimizer`
Gurobi	`http://www.gurobi.com`
Knitro	`http://www.ziena.com/knitro.htm`
Xpress	`http://www.solver.com/xlsxpresseng.htm`
Minos	`http://www.sbsi-sol-optimize.com/asp/sol_product_minos.htm`
Lindo	`http://www.lindo.com`
LP-Solve	`http://lpsolve.sourceforge.net/5.5/Intro.htm`
Mosek	`http://www.mosek.com/`

Tabla B.2: Listado de *solver* para la resolución de modelos de optimización.

mencionados en la Tabla B.1, aunque también es común que ofrezcan diversas API para crear los modelos empleando lenguajes de programación como C/C++, Java, Python o lenguajes de la plataforma .NET.

No todos los *solver* presentados en la Tabla B.1 son capaces de soportar cualquier tipo de modelo (existen modelos lineales, lineales enteros, no lineales, etc.). En la Tabla B.3 se muestran los principales modelos soportados por cada *solver* donde LP (del inglés, *Linear Programming*) representa a modelos lineales, MIP (del inglés, *Mixed-Integer Programming*) representa a modelos lineales mixtos, QP (del inglés, *Quadratic Programming*) representa a modelos cuadráticos, MIQP (del inglés, *Mixed-Integer Quadratic Programming*) representa a modelos cuadráticos mixtos y NLP (del inglés, *NonLinear Programming*) representa a modelos no lineales.

Solver	**Modelos soportados**				
	LP	MIP	QP	MIQP	NLP
CPLEX	✓	✓	✓	✓	
Gurobi	✓	✓	✓	✓	
Knitro	✓	✓	✓	✓	✓
Lindo	✓	✓	✓	✓	✓
LP-Solve	✓	✓			
Minos	✓	✓	✓	✓	✓
Mosek	✓	✓	✓	✓	✓
Xpress	✓	✓	✓	✓	

Tabla B.3: Principales modelos soportados por cada *solver* de la Tabla B.2 (información extraída de `http://www.maximal-usa.com/solvers/`).

Apéndice C

Relación de instancias empleadas en la experimentación

En este apéndice se presenta la relación de las instancias empleadas en el Capítulo 5 para la evaluación de los algoritmos exactos y heurísticos propuestos. Éstas están divididas en seis conjuntos: «Small-1», «Grid-1», «Harwell-Boeing-1», «Small-2», «Grid-2» y «Harwell-Boeing-2», mostrados respectivamente en las tablas C.1, C.2, C.3, C.4, C.5 y C.6.

- «Small-1» (42 instancias)

«**Small-1**»					
p17_16_24	p28_17_18	p41_19_20	p53_20_22	p69_21_23	p85_23_26
p18_16_21	p29_17_18	p44_19_25	p54_20_28	p72_22_49	p86_23_24
p19_16_19	p31_18_21	p46_19_20	p55_20_24	p76_22_30	p91_24_33
p20_16_18	p35_18_19	p48_19_21	p61_21_22	p77_22_37	p92_24_26
p25_17_20	p37_18_20	p50_19_25	p63_21_42	p78_22_31	p98_24_29
p26_17_19	p38_18_19	p51_20_28	p64_21_22	p81_23_46	p99_24_27
p27_17_19	p39_18_19	p52_20_27	p67_21_22	p83_23_24	p100_24_34

Tabla C.1: Relación de instancias del conjunto «Small-1».

- «Grid-1» (36 instancias)

«**Grid-1**»					
Grid3x3	Grid3x9	Grid4x8	Grid5x8	Grid6x9	Grid8x8
Grid3x4	Grid3x10	Grid4x9	Grid5x9	Grid6x10	Grid8x9
Grid3x5	Grid4x4	Grid4x10	Grid5x10	Grid7x7	Grid8x10
Grid3x6	Grid4x5	Grid5x5	Grid6x6	Grid7x8	Grid9x9
Grid3x7	Grid4x6	Grid5x6	Grid6x7	Grid7x9	Grid9x10
Grid3x8	Grid4x7	Grid5x7	Grid6x8	Grid7x10	Grid10x10

Tabla C.2: Relación de instancias del conjunto «Grid-1».

- «Harwell-Boeing-1» (34 instancias)

«**Harwell-Boeing-1**»					
arc130	bcsstk02	curtis54	ibm32	mcca	west0156
ash85	bcsstk04	dwt_234	impcol_b	nos1	west0167
bcspwr01	bcsstk05	fs_183_1	impcol_c	nos4	will199
bcspwr02	bcsstk22	gent113	lns_131	pores_1	will57
bcspwr03	can_144	gre_115	lund_a	steam3	
bcsstk01	can_161	gre_185	lund_b	west0132	

Tabla C.3: Relación de instancias del conjunto «Harwell-Boeing-1».

- «Small-2» (84 instancias)

«**Small-2**»					
p17_16_24	p31_18_21	p45_19_25	p59_20_23	p73_22_29	p87_23_30
p18_16_21	p32_18_20	p46_19_20	p60_20_22	p74_22_30	p88_23_26
p19_16_19	p33_18_21	p47_19_21	p61_21_22	p75_22_25	p89_23_27
p20_16_18	p34_18_21	p48_19_21	p62_21_30	p76_22_30	p90_23_35
p21_17_20	p35_18_19	p49_19_22	p63_21_42	p77_22_37	p91_24_33
p22_17_19	p36_18_20	p50_19_25	p64_21_22	p78_22_31	p92_24_26
p23_17_23	p37_18_20	p51_20_28	p65_21_24	p79_22_29	p93_24_27
p24_17_29	p38_18_19	p52_20_27	p66_21_28	p80_22_30	p94_24_31
p25_17_20	p39_18_19	p53_20_22	p67_21_22	p81_23_46	p95_24_27
p26_17_19	p40_18_32	p54_20_28	p68_21_27	p82_23_24	p96_24_27
p27_17_19	p41_19_20	p55_20_24	p69_21_23	p83_23_24	p97_24_26
p28_17_18	p42_19_24	p56_20_23	p70_21_25	p84_23_26	p98_24_29
p29_17_18	p43_19_22	p57_20_24	p71_22_29	p85_23_26	p99_24_27
p30_17_19	p44_19_25	p58_20_21	p72_22_49	p86_23_24	p100_24_34

Tabla C.4: Relación de instancias del conjunto «Small-2».

- «Grid-2» (81 instancias)

«**Grid-2**»					
Grid3x3	Grid6x18	Grid12x6	Grid15x21	Grid21x9	Grid24x24
Grid3x6	Grid6x21	Grid12x9	Grid15x24	Grid21x12	Grid24x27
Grid3x9	Grid6x24	Grid12x12	Grid15x27	Grid21x15	Grid27x3
Grid3x12	Grid6x27	Grid12x15	Grid18x3	Grid21x18	Grid27x6
Grid3x15	Grid9x3	Grid12x18	Grid18x6	Grid21x21	Grid27x9
Grid3x18	Grid9x6	Grid12x21	Grid18x9	Grid21x24	Grid27x12
Grid3x21	Grid9x9	Grid12x24	Grid18x12	Grid21x27	Grid27x15
Grid3x24	Grid9x12	Grid12x27	Grid18x15	Grid24x3	Grid27x18
Grid3x27	Grid9x15	Grid15x3	Grid18x18	Grid24x6	Grid27x21
Grid6x3	Grid9x18	Grid15x6	Grid18x21	Grid24x9	Grid27x24
Grid6x6	Grid9x21	Grid15x9	Grid18x24	Grid24x12	Grid27x27
Grid6x9	Grid9x24	Grid15x12	Grid18x27	Grid24x15	
Grid6x12	Grid9x27	Grid15x15	Grid21x3	Grid24x18	
Grid6x15	Grid12x3	Grid15x18	Grid21x6	Grid24x21	

Tabla C.5: Relación de instancias del conjunto «Grid-2».

- «Harwell-Boeing-2» (87 instancias)

«**Harwell-Boeing-2**»					
494_bus	bcsstk06	dwt__419	impcol_b	nos5	str__200
662_bus	bcsstk20	dwt__503	impcol_c	nos6	str__600
685_bus	bcsstk22	dwt__592	impcol_d	plat362	str____0
arc130	bcsstm07	fs_183_1	impcol_e	plskz362	west0132
ash292	can__144	fs_541_1	lns__131	pores_1	west0156
ash85	can__161	fs_680_1	lns__511	pores_3	west0167
bcspwr01	can__292	gent113	lund_a	saylr1	west0381
bcspwr02	can__445	gre_216a	lund_b	saylr3	west0479
bcspwr03	curtis54	gre__115	mbeacxc	sherman4	west0497
bcspwr04	dwt__209	gre__185	mcca	shl__200	west0655
bcspwr05	dwt__221	gre__343	nnc261	shl__400	will199
bcsstk01	dwt__234	gre__512	nnc666	shl____0	will57
bcsstk02	dwt__245	hor__131	nos1	steam1	
bcsstk04	dwt__310	ibm32	nos2	steam2	
bcsstk05	dwt__361	impcol_a	nos4	steam3	

Tabla C.6: Relación de instancias del conjunto «Harwell-Boeing-2».

Apéndice D

Resultados para el conjunto de instancias «Harwell-Boeing»

En este apéndice se presentan las mejores cotas inferiores y superiores encontradas de para las instancias del conjunto «Harwell-Boeing-1» (ver Apéndice C) empleando los algoritmos propuestos en el Capítulo 3. Los resultados son mostrados en la Tabla D.1, donde LB representa a la mejor cota inferior y UB representa a la mejor cota superior para cada instancia. Además, se reportan también el número de vértices (n) y el número de aristas (m) de cada instancia.

«**Harwell-Boeing-1**»									
	n	***m***	***LB***	***UB***		***n***	***m***	***LB***	***UB***
pores_1	30	103	17	17	lns__131	123	275	6	30
ibm32	32	90	20	23	arc130	130	715	62	202
bcspwr01	39	46	5	5	bcsstk04	132	1758	107	310
bcsstk01	48	176	22	32	west0132	132	404	14	71
bcspwr02	49	59	5	5	impcol_c	137	352	11	46
curtis54	54	124	10	13	can__144	144	576	25	25
will57	57	127	7	11	lund_a	147	1151	37	113
impcol_b	59	281	18	55	lund_b	147	1147	37	111
bcsstk02	66	2145	1089	1089	bcsstk05	153	1135	34	115
steam3	80	424	20	20	west0156	156	371	12	56
ash85	85	219	10	16	nos1	158	312	4	4
nos4	100	247	12	12	can__161	161	608	21	52
gent113	104	549	19	87	west0167	167	489	12	55
bcsstk22	110	254	6	13	mcca	168	1662	58	390
gre__115	115	267	10	36	fs_183_1	183	701	52	190
dwt__234	117	162	6	12	gre__185	185	650	19	48
bcspwr03	118	179	5	10	will199	199	660	16	132

Tabla D.1: Mejores LB y UB conocidos para las instancias del conjunto «Harwell-Boeing-1».

Además de los resultados anteriores, en la Tabla D.2 se presentan los mejores valores encontrados para las instancias del conjunto «Harwell-Boeing-2» (ver Apéndice C) empleando los algoritmos heurísticos propuestos en el Capítulo 4.

«**Harwell-Boeing-2**»							
	n	***m***	***mejor***		***n***	***m***	***mejor***
pores_1	30	103	17	ash292	292	958	36
ibm32	32	90	23	can__292	292	1124	96
bcspwr01	39	46	5	dwt__310	310	1069	26
bcsstk01	48	176	32	gre__343	343	1092	48
bcspwr02	49	59	5	dwt__361	361	1296	39
curtis54	54	124	13	plat362	362	2712	155
will57	57	127	11	plskz362	362	880	30
impcol_b	59	281	55	str____0	363	3049	560
bcsstk02	66	2145	1089	str__200	363	3244	606
steam3	80	424	20	str__600	363	2446	388
ash85	85	219	16	west0381	381	2150	480
nos4	100	247	12	dwt__419	419	1572	55
gent113	104	549	87	bcsstk06	420	3720	227
bcsstk22	110	254	13	bcsstm07	420	3416	199
gre__115	115	267	36	impcol_d	425	1267	64
dwt__234	117	162	12	hor__131	434	2138	145
bcspwr03	118	179	10	bcspwr05	443	590	18
lns__131	123	275	30	can__445	445	1682	123
arc130	130	715	202	pores_3	456	1769	29
bcsstk04	132	1758	310	bcsstk20	467	1295	20
west0132	132	404	71	nos5	468	2352	193
impcol_c	137	352	46	west0479	479	1889	287
can__144	144	576	25	mbeacxc	487	41686	16329
lund_a	147	1151	113	494_bus	494	586	17
lund_b	147	1147	111	west0497	497	1715	181
bcsstk05	153	1135	115	dwt__503	503	2762	138
west0156	156	371	56	lns__511	503	1425	71
nos1	158	312	4	gre__512	512	1680	72
can__161	161	608	52	fs_541_1	541	2466	296
west0167	167	489	55	sherman4	546	1341	36
mcca	168	1662	390	dwt__592	592	2256	70
fs_183_1	183	701	190	steam2	600	6580	308
gre__185	185	650	48	nos2	638	1272	4
will199	199	660	132	west0655	655	2841	466
impcol_a	206	557	47	662_bus	662	906	27
dwt__209	209	767	58	shl____0	663	1720	387
gre_216a	216	660	52	shl__200	663	1709	385
dwt__221	221	704	27	shl__400	663	1682	357
impcol_e	225	1187	170	nnc666	666	2148	80
saylr1	238	445	16	nos6	675	1290	29
steam1	240	1761	182	fs_680_1	680	1464	17
dwt__245	245	608	27	saylr3	681	1373	45
nnc261	261	794	46	685_bus	685	1282	35
bcspwr04	274	669	29				

Tabla D.2: Mejores valores encontrados para las instancias del conjunto «Harwell-Boeing-2».

Índice alfabético

W,

Bibliografía

[1] Adolphson, D. y T. C. Hu: *Optimal linear ordering.* SIAM Journal on Applied Mathematics, 25(3):403–423, 1973.

[2] Andrade, D. V. y M. G. C. Resende: *GRASP with Evolutionary Path-Relinking.* En *Seventh Metaheuristics International Conference (MIC)*, 2007.

[3] Andrade, D. V. y M. G. C. Resende: *GRASP with Path-Relinking for Network Migration Scheduling.* En *Proceedings of International Network Optimization Conference (INOC)*, 2007.

[4] Arora, S., A. Frieze y H. Kaplan: *A new rounding procedure for the assignment problem with applications to dense graph arrangement problems.* En *37th Annual IEEE Symposium on Foundations of Computer Science (FOCS'96)*, volumen 0, páginas 21–30, Los Alamitos, CA, USA, 1996. IEEE Computer Society, ISBN 0-8186-7594-2.

[5] Arora, S., A. Frieze y H. Kaplan: *A new rounding procedure for the assignment problem with applications to dense graph arrangement problems.* Mathematical Programming, 92:1–36, 2002, ISSN 0025-5610.

[6] Arora, S., D. Karger y M. Karpinski: *Polynomial Time Approximation Schemes for dense instances of NP-hard problems.* En *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC'95, páginas 284–293, New York, NY, USA, 1995. ACM, ISBN 0-89791-718-9.

[7] Arora, S., D. Karger y M. Karpinski: *Polynomial Time Approximation Schemes for dense instances of NP-hard problems.* Journal of Computer and System Sciences, 58:193–210, 1999, ISSN 0022-0000.

[8] Ausiello, G., M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi y V. Kann: *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999, ISBN 3540-65431-3.

[9] Barth, D.: *Bandwidth and Cutwidth of the Mesh of d-Ary Trees.* En *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*, Euro-Par'96, páginas 243–246, London, UK, 1996. Springer-Verlag, ISBN 3-540-61626-8.

[10] Barth, D., F. Pellegrini, A. Raspaud y J. Roman: *On Bandwidth, Cutwidth, and Quotient Graphs.* En *RAIRO Informatique Théorique et Applications*, número 29, páginas 487–508, 1995.

[11] Bellman, R.: *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[12] Berend, D., E. Korach y V. Lipets: *Minimal Cutwidth Linear Arrangements of Abelian Cayley graphs.* Discrete Mathematics, 308(20):4670 – 4695, 2008, ISSN 0012-365X.

[13] Berthomé, P. y N. Nisse: *A unified FPT Algorithm for Width of Partition Functions*. Informe técnico RR-6646, INRIA, 2008. `http://hal.inria.fr/inria-00321766/PDF/RR-6646.pdf`.

[14] Bertsekas, D. P.: *Network Optimization: Continuous and Discrete Models*. Athena Scientific, Belmont, MA, 1998.

[15] Bezrukov, S. L., J. D. Chavez, L. H. Harper, M. Röttger y U. P. Schroeder: *The congestion of n-cube layout on a rectangular grid*. Discrete Mathematics, 213:13–19, 2000, ISSN 0012-365X.

[16] Birattari, M., L. Paquete, T. Stützle y K. Varrentrapp: *Classification of Metaheuristics and Design of Experiments for the Analysis of Components*. Informe técnico, Darmstadt University of Technology, Germany, 2001.

[17] Blin, G., G. Fertin, D. Hermelin y S. Vialette: *Fixed-parameter algorithms for protein similarity search under mRNA structure constraints*. Journal of Discrete Algorithms, 6:618–626, 2008, ISSN 1570-8667.

[18] Blum, C. y A. Roli: *Metaheuristics in combinatorial optimization: Overview and conceptual comparison*. ACM Computing Surveys (CSUR), 35(3):268–308, 2003, ISSN 0360-0300.

[19] Blum, C., A. Roli y E. Alba: *An introduction to metaheuristic techniques*. En *Parallel Metaheuristics: A New Class of Algorithms*, página 3–42. John Wiley, 2005.

[20] Bodlaender, H. L., M. R. Fellows y D. M. Thilikos: *Derivation of Algorithms for Cutwidth and Related Graph Layout Parameters*. Informe técnico UU-CS, (Ext. rep. 2002-032), Utrecht, The Netherlands: Utrecht University. Information and Computing Sciences, 2002.

[21] Bodlaender, H. L., F. Fomin, A. Koster, D. Kratsch y D. Thilikos: *On Exact Algorithms for Treewidth*. En *Algorithms – ESA 2006*, volumen 4168 de *Lecture Notes in Computer Science*, páginas 672–683. Springer Berlin / Heidelberg, 2006.

[22] Bodlaender, H. L., Fellows M. R. y Thilikos D. M.: *Derivation of algorithms for Cutwidth and related graph layout parameters*. Journal of Computer and System Sciences, 75:231–244, 2009, ISSN 0022-0000.

[23] Boese, K.: *A new Adaptive Multi-Start technique for combinatorial global optimizations*. Operations Research Letters, 16(2):101–113, 1994, ISSN 01676377.

[24] Booth, H. D., R. Govindan, M. A. Langston y S. Ramachandramurthi: *Cutwidth approximation in linear time*. En *Proceedings of the Second Great Lakes Symposium on VLSI*, páginas 70–73, 1992.

[25] Botafogo, R. A.: *Cluster analysis for hypertext systems*. En *16th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, páginas 116–125, 1993.

[26] Cerný, V.: *Thermodynamical approach to the Traveling Salesman Problem: An efficient simulation algorithm*. Journal of Optimization Theory and Applications, 45(1):41–51, 1985.

[27] Chao, L. F. y E.H. M. Sha: *Algorithms for Min-Cut Linear Arrangements of outerplanar graphs*. En *IEEE International Symposium on Circuits and Systems, ISCAS'92*, volumen 4, páginas 1851–1854, 1992.

[28] Chavez, J. D. y R. Trapp: *The Cyclic Cutwidth of trees.* Discrete Applied Mathematics, 87(1-3):25–32, 1998, ISSN 0166-218X.

[29] Chen, M. H. y S. L. Lee: *Linear time algorithms for k-cutwidth problem.* En *Algorithms and Computation*, volumen 650 de *Lecture Notes in Computer Science*, páginas 21–30. Springer Berlin / Heidelberg, 1992.

[30] Chong, E. K. P. y S. H. Zak: *An introduction to Optimization.* John Wiley & Sons, Inc., 2008, ISBN 0-471-75800-0.

[31] Chung, F. R. y P. D. Seymour: *Graphs with small Bandwidth and Cutwidth.* Discrete Mathematics, 75:113–119, 1989, ISSN 0012-365X.

[32] Chung, F. R. K.: *Some problems and results on labelings of graphs.* The Theory and Applications of Graphs, páginas 255–264, 1981. (ed. G. Chartrand), John Wiley and Sons.

[33] Chung, F. R. K.: *On the Cutwidth and the Topological Bandwidth of a tree.* SIAM Journal on Algebraic Discrete Methods, 6:268–277, 1985.

[34] Chung, F. R. K.: *Labelings of graphs. Selected Topics in Graph Theory*, volumen 3, capítulo 7, páginas 151–168. Academic Press, San Diego, CA, 1988.

[35] Chung, M. J., F. Makedon, I. H. Sudborough y J. Turner: *Polynomial time algorithms for the MIN CUT problem on degree restricted trees.* En *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, páginas 262–271, Washington, DC, USA, 1982. IEEE Computer Society.

[36] Chung, M. J., F. Makedon, I. H. Sudborough y J. Turner: *Polynomial time algorithms for the min cut problem on degree restricted trees.* SIAM Journal on Computing, 14:158–177, 1985, ISSN 0097-5397.

[37] Cohoon, J. y S. Sahni: *Exact algorithms for the Board Permutation Problem.* Informe técnico, Department of Computer Science. University of Minnesota, 1982.

[38] Cohoon, J. y S. Sahni: *Heuristics for the Board Permutation Problem.* Informe técnico, Department of Computer Science. University of Minnesota, 1982.

[39] Cohoon, J. y S. Sahni: *Exact algorithms for special cases of the Board Permutation Problem.* En *Proceedings of the Allerton Conference on Communication, Control and Computing*, páginas 246–255, 1983.

[40] Cohoon, J. y S. Sahni: *Heuristics for Backplane Ordering.* Journal of VLSI and Computer Systems, 2:37–61, 1987.

[41] Cook, S. A.: *The complexity of theorem-proving procedures.* En *Proceedings of the third annual ACM symposium on Theory of Computing*, STOC '71, páginas 151–158, New York, NY, USA, 1971. ACM.

[42] Cormen, T. H., C. E. Leiserson, R. L. Rivest y Stein C.: *Introduction to Algorithms.* MIT Press, Cambridge, MA, spanish2nd edición, 2001, ISBN 0-262-03293-7.

[43] Dantzig, G. B.: *Maximization of a linear function of variables subject to linear inequalities.* En *Activity Analysis of Production and Allocation.* Wiley, 1951.

[44] Díaz, J., A. Gibbons, G. E. Pantziou, M. J. Serna, P. G. Spirakis y J. Toran: *Parallel algorithms for the minimum cut and the minimum length tree layout problems.* Theoretical Computer Science, 181:267–287, 1997, ISSN 0304-3975.

[45] Díaz, J., M.D. Penrose, J. Petit y M.J. Serna: *Approximating Layout Problems on Random Geometric Graphs.* Journal of Algorithms, 39:2001, 2001.

[46] Díaz, J., J. Petit y M. J. Serna: *A survey of graph layout problems.* ACM Computing Surveys (CSUR), 34(3):313–356, 2002, ISSN 0360-0300.

[47] Díaz, J., J. Petit, M.J. Serna y L. Trevisan: *Approximating Layout Problems on Random Sparse Graphs.* Informe técnico LSI-98-44-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, 1998. (Presented in the Fifth Czech-Slovak International Symposium on Combinatorics, Graph Theory, Algorithms and Applications).

[48] Díaz, J., J. Petit, M.J. Serna y L. Trevisan: *Approximating Layout Problems on Random Graphs.* Discrete Mathematics, 235:245–253, 2001, ISSN 0012-365X.

[49] Diks, K., H. Djidjev, O. Sýkora y I. Vrt'o: *Edge Separators of Planar and Outerplanar Graphs with Applications.* Journal of Algorithms, 14(2):258–279, 1993.

[50] Djidjev, H. N. y I. Vrt'o: *Crossing numbers and cutwidths.* Journal of Graph Algorithms and Applications, (7):245–251, 2003.

[51] Duarte, A., J. J. Pantrigo y M. Gallego: *Metaheurísticas.* Dykinson, 2007, ISBN 978-84-9849-016-9.

[52] Even, S. y Y. Shiloach: *NP-completeness of several arrangement problems.* Informe técnico TR-43, Department of Computer Science, Technion, Haifa, 1975.

[53] Fellows, M. R. y Langston M. A.: *Layout permutation problems and well-partially-ordered sets.* En *Proceedings of the fifth MIT conference on Advanced research in VLSI*, páginas 315–327, Cambridge, MA, USA, 1988. MIT Press.

[54] Fellows, M. R. y M. A. Langston: *On well-partial-order theory and its application to combinatorial problems of VLSI design.* SIAM Journal Discrete Mathematics, 5(1):117–126, 1992, ISSN 0895-4801.

[55] Feo, T. y M. G. C. Resende: *A probabilistic heuristic for a computationally difficult set covering problem.* Operations Research Letters, (8):67–71, 1989.

[56] Feo, T. y M. G. C. Resende: *Greedy Randomized Adaptive Search Procedures.* Journal of Global Optimization, (6):109–133, 1995.

[57] Floudas, C. A. y P. M. Pardalos (editores): *Encyclopedia of Optimization, Second Edition.* Springer, 2009, ISBN 978-0-387-74758-3.

[58] Ford, L. R. y D. R. Fulkerson: *Flows in Networks.* Princeton University Press, 1962.

[59] Gallego, M.: *Resolución exacta y aproximada del Problema de la Diversidad Máxima.* Tesis de Doctorado, Escuela Técnica Superior de Ingeniería Informática. Universidad Rey Juan Carlos., 2008.

[60] Gallego, M., F. Gortázar y E. G. Pardo: *Optsicom Optimization Suite, un conjunto de herramientas para la investigación en optimización.* En *VII Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados - MAEB (CEDI 2010)*, páginas 329–336, 2010.

[61] Garey, M. R., R. L. Graham, D. S. Johnson y D. E. Knuth: *Complexity Results for Bandwidth Minimization.* SIAM Journal on Applied Mathematics, 34(3):477–495, 1978, ISSN 00361399.

[62] Garey, M. R. y D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979, ISBN 0-7167-1044-7.

[63] Garey, M. R., D. S. Johnson y L. J. Stockmeyer: *Some Simplified NP-Complete Graph Problems*. Theoretical Computer Science, 1(3):237–267, 1976.

[64] Gavril, F.: *Some NP-Complete problems on graphs*. En *Proceedings of the Eleventh Conference on Information Sciences and Systems*, páginas 91–95, Baltimore, MD, 1977.

[65] Gendreau, M. y J. Y. Potvin: *Metaheuristics in Combinatorial Optimization*. Annals of Operations Research, 140(1):189–213, 2005.

[66] Gendreau, M. y J. Y. Potvin: *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*, volumen 146. Springer, spanish2nd edición, 2010.

[67] Glover, F.: *Heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(1):156—-166, 1977.

[68] Glover, F.: *Future paths for integer programming and links to artificial intelligence*. Computers and Operations Research, 13(5):533–549, 1986, ISSN 0305-0548.

[69] Glover, F.: *Tabu search for nonlinear and parametric optimization (with links to genetic algorithms)*. Discrete Applied Mathematics, 49(1-3):231–255, 1994, ISSN 0166-218X.

[70] Glover, F.: *Tabu search and adaptive memory programming – Advances, applications and challenges*. En *Interfaces in Computer Science and Operations Research*, páginas 1–75. Kluwer, 1996.

[71] Glover, F.: *A Template for Scatter Search and Path Relinking*. En *Selected Papers from the Third European Conference on Artificial Evolution*, volumen 1363, páginas 3–54, London, UK, 1997. Springer-Verlag, ISBN 3-540-64169-6.

[72] Glover, F. y G. A. Kochenberger: *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. Springer, 2003, ISBN 1402072635.

[73] Glover, F. y M. Laguna: *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997, ISBN 079239965X.

[74] Glover, F. y M. Laguna: *Fundamentals of Scatter Search and Path Relinking*. Control and Cybernetics, 29(3):653–684, 2000.

[75] Glover, F., M. Laguna y R. Martí: *Scatter Search and Path Relinking: foundations and advanced designs*. New optimization techniques in engineering, páginas 87–100, 2004.

[76] Gomory, R. E.: *Outline of an algorithm for integer solutions to linear programs*. Bulletin of the American Mathematical Society, 64(5):275–278, 1958.

[77] Gomory, R. E. y T. C. Hu: *Multi-Terminal Network Flows*. Journal of the Society for Industrial and Applied Mathematics, 9(4):551–570, 1961, ISSN 03684245.

[78] Greistorfer, P. y S. Voss: *Controlled pool maintenance in combinatorial optimization*. En *Conference on Adaptive Memory and Evolution: Tabu Search and Scatter Search*, 2001.

[79] Gurari, E. M. y I. H. Sudborough: *Cutwidth problems in graphs*. En *Proceedings of the Nineteenth Annual Allerton Conference on Communication, Control, and Computing*, páginas 752–761, 1981.

[80] Gurari, E. M. y I. H. Sudborough: *Improved dynamic programming algorithms for Bandwidth Minimization and the MinCut Linear Arrangement problem.* Journal of Algorithms, 5(4):531 – 546, 1984, ISSN 0196-6774.

[81] Gurski, F.: *Graph parameters measuring neighbourhoods in graphs-Bounds and applications.* Discrete Applied Mathematics, 156:1865–1874, 2008, ISSN 0166-218X.

[82] Hao, J.: *Maximum Cutwidth problem for graphs.* Applied Mathematics - A Journal of Chinese Universities, 18:235–242, 2003, ISSN 1005-1031.

[83] Harper, L. H.: *Optimal Assignments of Numbers to Vertices.* Journal of the Society for Industrial and Applied Mathematics, 12(1):131–135, 1964, ISSN 03684245.

[84] Harper, L. H.: *Optimal numberings and isoperimetric problems on graphs.* Journal of Combinatorial Theory, (1):385–393, 1966.

[85] Heggernes, P., D. Lokshtanov, R. Mihai y C. Papadopoulos: *Cutwidth of Split Graphs, Threshold Graphs, and Proper Interval Graphs.* En *Graph-Theoretic Concepts in Computer Science*, páginas 218–229. Springer-Verlag, Berlin, Heidelberg, 2008, ISBN 978-3-540-92247-6.

[86] Heggernes, P., P. Van't Hof, D. Lokshtanov y J. Nederlof: *Computing the cutwidth of bipartite permutation graphs in linear time.* En *Proceedings of the 36th international conference on Graph-theoretic concepts in computer science*, WG'10, páginas 75–87, Berlin, Heidelberg, 2010. Springer-Verlag, ISBN 3-642-16925-2, 978-3-642-16925-0.

[87] Hochbaum, D. S.: *Approximation algorithms for NP-hard problems.* PWS Publishing Co., Boston, MA, USA, 1997, ISBN 0-534-94968-1.

[88] Holland, J. H.: *Adaptation in natural and artificial systems.* The University of Michigan Press, 1975.

[89] Jünger, M. y P. Mutzel: *The polyhedral approach to the maximum planar subgraph problem: New chances for related problems.* En *Graph Drawing*, volumen 894 de *Lecture Notes in Computer Science*, páginas 119–130. Springer Berlin / Heidelberg, 1994.

[90] Jünger, M., G. Reinelt y G. Rinaldi: *The Traveling Salesman Problem.* En *Handbook on Operations Research and Management Science*, volumen 7, páginas 225–330. North-Holland, 1995.

[91] Jünger, M., G. Reinelt y S. Thienel: *Practical Problem Solving with Cutting Plane Algorithms in Combinatorial Optimization.* Combinatorial Optimization, DIMACS Series in Discrete Mathematics and Computer Science, 20:111–152, 1995.

[92] Juvan, M. y B. Mohar: *Optimal linear labelings and eigenvalues of graphs.* Discrete Applied Mathematics, 36:153–168, 1992, ISSN 0166-218X.

[93] Karger, D.R.: *A randomized fully Polynomial Time Approximation Scheme for the all-terminal network reliability problem.* SIAM Journal on Computing, 29(2):492–514, 1999.

[94] Karmarkar, N.: *A new polynomial-time algorithm for linear programming.* Combinatorica, 4:373–395, 1984, ISSN 0209-9683.

[95] Karp, R. M.: *Reducibility Among Combinatorial Problems.* En *Complexity of Computer Computations*, páginas 85–103. Plenum Press, 1972. Edited by Miller, R. E. and Thatcher, J. W.

[96] Kirkpatrick, S., C. D. Gelatt y M. P. Vecchi: *Optimization by Simulated Annealing*. Science, 220(4598):671–680, 1983.

[97] Kocay, W. y D. L. Kreher: *Graphs, Algorithms and Optimization*. Chapman & Hall/CRC, 2004, ISBN 1-58488-396-0.

[98] Korach, E. y N. Solel: *Tree-width, Path-width and Cutwidth*. Discrete Applied Mathematics, 43:97–101, 1993, ISSN 0166-218X.

[99] Laguna, M. y R. Martí: *A GRASP for coloring sparse graphs*. Computational Optimization and Applications, 19:165–178, 2001, ISSN 0926-6003.

[100] Laguna, M. y R. Martí: *Scatter Search: Methodology and Implementations in C*. Kluwer Academic Publishers, Boston, 2003, ISBN 1-58488-396-0.

[101] Land, A. H. y A. G. Doig: *An automatic method of solving discrete programming problems*. Econometrica, 3(28):497—-520, 1960.

[102] LaPaugh, A. S.: *Recontamination does not help to search a graph*. Journal of the ACM (JACM), 40:224–245, 1993, ISSN 0004-5411.

[103] Leighton, T. y S. Rao: *Multicommodity Max-Flow Min-Cut theorems and their use in designing approximation algorithms*. Journal of the ACM (JACM), 46:787–832, 1999, ISSN 0004-5411.

[104] Lemke, C. E.: *The dual method of solving the linear programming problem*. Nav. Res. Logist. Q., (1):36–47, 1954.

[105] Lengauer, T.: *Upper and lower bounds on the complexity of the Min-Cut linear arrangement problem on trees*. SIAM Journal on Algebraic and Discrete Methods, 3(1):99–113, 1982.

[106] Lin, W. L., A. H. Farrahi y M. Sarrafzadeh: *On the Power of Logic Resynthesis*. SIAM Journal on Computing, 29:1257–1289, 2000, ISSN 0097-5397.

[107] Lin, W. L. y M. Sarrafzadeh: *A linear arrangement problem with applications*. En *IEEE International Symposium on Circuits and Systems. ISCAS '95*, volumen 1, páginas 57–60, 1995.

[108] Lin, Y.: *Cutwidth and related parameters of graphs*. Journal of Zhengzhou University, Natural Science Edition, 1, 2002. (En chino).

[109] Lin, Y.: *The cutwidth of trees with diameter at most 4*. Applied Mathematics - A Journal of Chinese Universities, 18:361–369, 2003, ISSN 1005-1031.

[110] Lin, Y., X. Li y A. Yang: *A degree sequence method for the Cutwidth problem of graphs*. Applied Mathematics - A Journal of Chinese Universities, 17:125–134, 2002, ISSN 1005-1031.

[111] Lin, Y. y A. Yang: *On 3-Cutwidth critical graphs*. Discrete Mathematics, 275(1-3):339–346, 2004, ISSN 0012-365X.

[112] Lipets, V.: *Bounds on Mincut for Cayley Graphs over Abelian Groups*. Theory of Computing Systems, 45:372–380, 2009, ISSN 1432-4350.

[113] Liu, H. y J. Yuan: *Cutwidth problem on graphs*. Applied Mathematics, Ser. A. A Journal of Chinese Universities, 10:339–348, 1995. (En chino).

[114] Luttamaguzi, J., M. Pelsmajer, Z. Shen y B. Yang: *Integer programming solutions for several optimization problems in graph theory.* Informe técnico, Center for Discrete Mathematics and Theoretical Computer Science, DIMACS, 2005.

[115] Makedon, F., C. Papadimitriou y I. Sudborough: *Topological Bandwidth.* En *CAAP'83*, volumen 159 de *Lecture Notes in Computer Science*, páginas 317–331. Springer Berlin / Heidelberg, 1983.

[116] Makedon, F. y I. H. Sudborough: *Minimizing Width in Linear Layouts.* En *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, páginas 478–490, London, UK, 1983. Springer-Verlag, ISBN 3-540-12317-2.

[117] Makedon, F. y I. H. Sudborough: *On minimizing width in linear layouts.* Discrete Applied Mathematics, 23(3):243–265, 1989.

[118] Makedon, F. S., C. H. Papadimitriou y I. H. Sudborough: *Topological Bandwidth.* SIAM Journal on Algebraic and Discrete Methods, 6(3):418–444, 1985.

[119] Martí, R., V. Campos y E. Piñana: *A branch and bound algorithm for the Matrix Bandwidth minimization.* European Journal of Operational Research, 186(2):513–528, 2008.

[120] Martí, R. y M. Laguna: *Scatter Search: Diseño Básico y Estrategias avanzadas.* Revista Iberoamericana de Inteligencia Artificial, 19:123–130, 2003.

[121] Megiddo, N., S. L. Hakimi, M. R. Garey, D. S. Johnson y C. H. Papadimitriou: *The complexity of searching a graph.* Journal of the ACM (JACM), 35:18–44, January 1988, ISSN 0004-5411.

[122] Melián, B., J. A. Moreno y J. M. Moreno: *Metaheuristicas: Una visión global.* Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial, 19:7–28, 2003.

[123] Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller y E. Teller: *Equation of State Calculations by Fast Computing Machines.* Journal of Chemical Physics, 21:1087–1092, 1953.

[124] Miller, Z. y I. H. Sudborough: *A polynomial algorithm for recognizing small Cutwidth in hypergraphs.* En *VLSI Algorithms and Architectures*, volumen 227 de *Lecture Notes in Computer Science*, páginas 252–260. Springer Berlin / Heidelberg, 1986.

[125] Miller, Z. y I. H. Sudborough: *A polynomial algorithm for recognizing bounded cutwidth in hypergraphs.* Theory of Computing Systems, 24:11–40, 1991, ISSN 1432-4350.

[126] Mladenovic, N. y P. Hansen: *Variable Neighborhood Search.* Computers & Operations Research, 24(11):1097–1100, 1997.

[127] Mohar, B. y S. Poljak: *Eigenvalues in combinatorial optimization.* Combinatorial and Graph-Theoretical Problems in Linear Algebra, 50:107–151, 1993.

[128] Monien, B. y I. H. Sudborough: *Min Cut is NP-complete for edge weighted trees.* En *Automata, Languages and Programming*, volumen 226 de *Lecture Notes in Computer Science*, páginas 265–274. Springer Berlin / Heidelberg, 1986.

[129] Monien, B. y I. H. Sudborough: *Min cut is NP-complete for edge weighted trees.* Theoretical Computer Science, 58:209–229, 1988, ISSN 0304-3975.

[130] Monien, B. y I. Vrt'o: *Improved bounds on cutwidths of Shuffle-Exchange and de Bruijn graphs.* Parallel Processing Letters, 14(3-4):361–366, 2004.

[131] Moscato, P.: *On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts - Towards Memetic Algorithms*. Informe técnico, Caltech Concurrent Computation Program (Report 826). California Institute of Technology, 1989.

[132] Muradian, D.: *On Layout Problems of Kneser Graphs*. En *Computer Science and Information Technologies Conference*, páginas 341–342, Yerevan, Armenia., 2009.

[133] Muradjan, D. O. y T. È. Piliposjan: *Minimal numberings of vertices of a rectangular lattice*. Akademiya Nauk Armyanskoj SSR. Doklady., 70(1):21–27, 1980, ISSN 0321-1339. (En ruso).

[134] Muradjan, D. O. y T. È. Piliposjan: *The problem of finding the length and width of the complete p-partite graph*. Uchen. Zapiski Erevan. Gosunivers, (2):18–26, 1980. (En ruso).

[135] Mutzel, P.: *A Polyhedral Approach to Planar Augmentation and Related Problems*. En *ESA'95: Proceedings of the Third Annual European Symposium on Algorithms*, páginas 494–507, London, UK, 1995. Springer-Verlag, ISBN 3-540-60313-1.

[136] Nakano, K.: *Linear layouts of generalized hypercubes*. En *Graph-Theoretic Concepts in Computer Science*, volumen 790 de *Lecture Notes in Computer Science*, páginas 364–375. Springer Berlin / Heidelberg, 1994.

[137] Nakano, K.: *Linear Layout of Generalized Hypercubes*. International Journal on Foundations of Computer Science, 14(1):137–156, 2003.

[138] Nakano, K., W. Chen, T. Masuzawa, K. Hagihara y N. Tokura: *Cut-Width and Bisection Width of hypercube graph*. En *IEICE Transactions. J73-A(4)*, páginas 856–862, 1990. (En japonés).

[139] Oxford University, Oxford English Dictionary. Oxford University Press, 1989. Segunda edición.

[140] Pantrigo, J. J., R. Martí, A. Duarte y E. G. Pardo: *The Cutwidth Minimization Problem*. ALIO/INFORMS Joint International Meeting, 2010.

[141] Papadimitriou, C.: *The NP-Completeness of the bandwidth minimization problem*. Computing, 16:263–270, 1976, ISSN 0010-485X.

[142] Papadimitriou, C. H. y K. Steiglitz: *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998, ISBN 0-13-152462-3.

[143] Pardo, E. G., A. Duarte y J. J. Pantrigo: *Búsqueda Dispersa para el problema de Ordenación Lineal de Corte Mínimo*. En *VII Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados - MAEB (CEDI 2010)*, páginas 85–92, 2010, ISBN 978-84-92812-58-5.

[144] Parsons, T.: *Pursuit-evasion in a graph*. En *Theory and Applications of Graphs*, volumen 642 de *Lecture Notes in Mathematics*, páginas 426–441. Springer Berlin / Heidelberg, 1978.

[145] Penrose, M. D.: *Vertex Ordering and Partitioning Problems for Random Spatial Graphs*. The Annals of Applied Probability, 10(2):517–538, 2000, ISSN 10505164.

[146] Petit, J.: *Experiments on the Minimum Linear Arrangement Problem*. ACM Journal of Experimental Algorithmics, 8, 2003, ISSN 1084-6654.

[147] Piñana, E., I. Plana, V. Campos y R. Martí: *GRASP and Path Relinking for the Matrix Bandwidth Minimization*. European Journal of Operational Research, 153(1):200 – 210, 2004, ISSN 0377-2217.

[148] Pólya, G.: *How to Solve It.* Princeton University Press, spanish2nd edición, 1957.

[149] Raspaud, A., H. Schröder, O. Sýkora, L. Torok y I. Vrt'o: *Antibandwidth and cyclic antibandwidth of meshes and hypercubes.* Discrete Mathematics, 309(11):3541–3552, 2009, ISSN 0012-365X. 7th International Colloquium on Graph Theory - ICGT 05.

[150] Raspaud, A., O. Sýkora y I. Vrt'o: *Cutwidth of the de Bruijn graph.* Theoretical Informatics and Applications - RAIRO, (26):509–514, 1995.

[151] Raspaud, A., O. Sýkora y I. Vrt'o: *Congestion and Dilation, Similarities and Differences: a Survey.* En *Proceedings of the 7th International Colloquium on Structural Information and Communication Complexity*, páginas 269–280, 2000.

[152] Real Academia Española, Diccionario de la Lengua Española. Espasa-Calpe, 2001. Vigésima segunda edición.

[153] Resende, M. G. C. y D. V. Andrade: *Method and System for Network Migration Scheduling.* United States Patent Application Publication. US2009/0168665, 2009.

[154] Resende, M. G. C. y J. L. González-Velarde: *GRASP: Procedimientos de búsquedas miopes aleatorizados y adaptativos.* Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial, 19:61–76, 2003.

[155] Resende, M. G. C., R. Martí, M. Gallego y A. Duarte: *GRASP and path relinking for the max-min diversity problem.* Computers and Operations Research, 37:498–508, 2010, ISSN 0305-0548.

[156] Resende, M. G. C. y C. C. Ribeiro: *Greedy Randomized Adaptive Search Procedures.* En *Handbook of Metaheuristics*, páginas 219–249. F. Glover and G. Kochenberger, eds., Kluwer Academic Publishers, 2003.

[157] Resende, M. G. C. y R. F. Werneck: *A Hybrid Heuristic for the p-Median Problem.* Journal of Heuristics, 10:59–88, 2004, ISSN 1381-1231.

[158] Rios, F.: *Complete Graphs as a First Step Toward Finding the Cyclic Cutwidth of the n-Cube.* Informe técnico, California State University, San Bernardino McNair, 1996. Scholar's Program Summer Research Journal.

[159] Rolim, J., O. Sýkora y I. Vrt'o: *Optimal cutwidths and bisection widths of 2- and 3-dimensional meshes.* En *Graph-Theoretic Concepts in Computer Science*, volumen 1017 de *Lecture Notes in Computer Science*, páginas 252–264. Springer Berlin / Heidelberg, 1995.

[160] Rosing, K. E. y M. J. Hodgson: *Heuristic concentration for the p-median: an example demonstrating how and why it works.* Computers and Operations Research, 29:1317–1330, 2002, ISSN 0305-0548.

[161] Rosing, K. E. y C. S. ReVelle: *Heuristic concentration: Two stage solution construction.* European Journal of Operational Research, 97(1):75–86, 1997.

[162] Rutman, R.A.: *An algorithm for placement of interconnected elements based on minimum wire length.* En *Proceedings of the spring joint computer conference*, AFIPS '64 (Spring), páginas 477–491, New York, NY, USA, 1964. ACM.

[163] Schröder, H., O. Sýkora y I. Vrt'o: *Cyclic Cutwidth of the Mesh.* En *SOFSEM'99: Theory and Practice of Informatics*, volumen 1725 de *Lecture Notes in Computer Science*, páginas 761–761. Springer Berlin / Heidelberg, 1999.

[164] Schröder, H., O. Sýkora y I. Vrt'o: *Cyclic Cutwidths of the two-dimensional ordinary and cylindrical meshes.* Discrete Applied Mathematics, 143:123–129, 2004, ISSN 0166-218X.

[165] Shahrokhi, F., O. Sýkora, L. A. Székely y I. Vrt'o: *On Bipartite Drawings and the Linear Arrangement Problem.* SIAM Journal on Computing, 30(6):1773–1789, 2001, ISSN 0097-5397.

[166] Simonson, S.: *A variation on the Min Cut linear arrangement problem.* Theory of Computing Systems, 20:235–252, 1987, ISSN 1432-4350.

[167] Skodinis, K.: *Computing Optimal Linear Layouts of Trees in Linear Time.* En *Proceedings of the 8th Annual European Symposium on Algorithms*, ESA'00, páginas 403–414, London, UK, 2000. Springer-Verlag, ISBN 3-540-41004-X.

[168] Sýkora, O. y I. Vrt'o: *Edge separators for graphs of bounded genus with applications.* En *Graph-Theoretic Concepts in Computer Science*, volumen 570 de *Lecture Notes in Computer Science*, páginas 159–168. Springer Berlin / Heidelberg, 1992.

[169] Sýkora, O. y I. Vrt'o: *Edge Separators for Graphs of Bounded Genus with Applications.* Theoretical Computer Science, 112(2):419–429, 1993.

[170] Steinberg, L.: *The Backboard Wiring Problem: A Placement Algorithm.* SIAM Review, 3(1):37–50, 1961, ISSN 00361445.

[171] Takagi, K. y N. Takagi: *Minimum Cut Linear Arrangement of p-q Dags for VLSI Layout of Adder Trees.* IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, E82-A(5):767–774, 1999.

[172] Talbi, E G.: *Metaheuristics: from design to implementation.* Wiley, USA, 2009.

[173] Thilikos, D. M., M. J. Serna y H. L. Bodlaender: *A constructive linear time algorithm for small Cutwidth.* Informe técnico UU-CS (Ext. rep. 2000-24), Utrecht, The Netherlands: Utrecht University. Information and Computing Sciences, 2000.

[174] Thilikos, D. M., M. J. Serna y H. L. Bodlaender: *Constructive Linear Time Algorithms for Small Cutwidth and Carving-Width.* En *Proceedings of the 11th International Conference on Algorithms and Computation*, ISAAC'00, páginas 192–203, London, UK, 2000. Springer-Verlag, ISBN 3-540-41255-7.

[175] Thilikos, D. M., M. J. Serna y H. L. Bodlaender: *A polynomial algorithm for the Cutwidth of bounded degree graphs with small treewidth.* Informe técnico UU-CS (Ext. rep. 2001-04), Utrecht, The Netherlands: Utrecht University. Information and Computing Sciences, 2001.

[176] Thilikos, D. M., M. J. Serna y H. L. Bodlaender: *A Polynomial Time Algorithm for the Cutwidth of Bounded Degree Graphs with Small Treewidth.* En *Proceedings of the 9th Annual European Symposium on Algorithms*, ESA'01, páginas 380–390, London, UK, 2001. Springer-Verlag, ISBN 3-540-42493-8.

[177] Thilikos, D. M., M. J. Serna y H. L. Bodlaender: *Cutwidth I: A linear time fixed parameter algorithm.* Journal of Algorithms, 56(1):1–24, 2005, ISSN 0196-6774.

[178] Thilikos, D. M., M. J. Serna y H. L. Bodlaender: *Cutwidth II: Algorithms for partial w-trees of bounded degree,.* Journal of Algorithms, 56(1):25–49, 2005, ISSN 0196-6774.

[179] Vazirani, V. V.: *Approximation Algorithms.* Springer-Verlag, Berlin, 2001.

[180] Vrt'o, I.: *Cutwidth of the mesh of d-ary trees.* En *Euro-Par'97 Parallel Processing*, volumen 1300 de *Lecture Notes in Computer Science*, páginas 242–245. Springer Berlin / Heidelberg, 1997.

[181] Vrt'o, I.: *Cutwidth of the d-dimensional mesh of r-ary trees.* RAIRO Informatique Théorique et Applications, 34(6):515–519, 2000.

[182] Wada, K., H. Suzuki y K. Kawaguchi: *The crossing number of hypercube graphs.* En *Proceedings of the 43rd Convention of IPS Japan*, páginas 1–95, 1991. (En japonés).

[183] Wang, D., E. Clarke, Y. Zhu y J. Kukula: *Using Cutwidth to Improve Symbolic Simulation and Boolean Satisfiability.* En *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop*, HLDVT '01, páginas 165–170, Washington, DC, USA, 2001. IEEE Computer Society, ISBN 0-7695-1411-1.

[184] Yagiura, M. y T. Ibaraki: *On metaheuristic algorithms for combinatorial optimization problems.* The Transactions of the Institute of Electronics, Information and Communication Engineers, 2:83–1, 2001.

[185] Yannakakis, M.: *A polynomial algorithm for the MIN CUT linear arrangement of trees.* Annual IEEE Symposium on Foundations of Computer Science, 0:274–281, 1983, ISSN 0272-5428.

[186] Yannakakis, M.: *A polynomial algorithm for the Min-Cut linear arrangement of trees.* Journal of the ACM (JACM), 32:950–988, 1985, ISSN 0004-5411.

[187] Yuan, J. y S. Zhou: *Optimal labelling of unit interval graphs.* Applied Mathematics - A Journal of Chinese Universities, 10:337–344, 1995, ISSN 1005-1031.