Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Un algoritmo *branch and cut* para el problema de *mapping*

Tesis de Licenciatura en Ciencias de la Computación

Javier Marenco

Director: Lic. Irene Loiseau
Codirector: Lic. Pablo E. Coll
Buenos Aires, 1999

Agradecimientos

En primer lugar, quiero agradecer a Irene Loiseau por su constante apoyo y estímulo a lo largo de todo este trabajo. Sus indicaciones y sugerencias constituyeron un muy valioso aporte, sin el cual hubiera sido imposible completarlo. Muchas gracias!

Debo también agradecer a Pablo Coll, por varios motivos. En primer lugar, su ayuda sobre el entorno ABACUS fue fundamental, y sin su intervención nunca habría podido comenzar la implementación. Por otra parte, generosamente mantuvo conversaciones conmigo sobre este tema, de las cuales surgieron ideas que se incorporaron al algoritmo o que ayudaron en muchas demostraciones.

Quiero agradecer a los jurados, Isabel Méndez Díaz y Esteban Feuerstein, por sus comentarios sobre el trabajo, que fueron de gran ayuda y permitieron mejorar notablemente la redacción final. Además, Isabel Méndez Díaz dedicó una mañana a escuchar una presentación informal de este trabajo, de la cual surgieron valiosos comentarios y aclaraciones. Por otra parte, debo agradecer las sugerencias de Esteban Feuerstein sobre bibliografía relacionada con el tema.

Quiero agradecer también a Alejandro Hoese, Guillermo Durán y Cid Carvalho de Souza por haber prestado parte de su tiempo para pensar sobre el problema. Fue muy grato contar con su colaboración. Por otra parte, quiero agradecer a Alejandro Strejilevich la ayuda que me brindó con artículos sobre variaciones del problema.

Aunque no reciban este agradecimiento en forma directa, también debo mencionar a Max Böhm, Lisa Magnano-Bleehen y François Pellegrini por su ayuda para trabajar con ABACUS y localizar distintos artículos.

Por último, quiero agraceder también a Alejandra Davidson, Eduardo Arias, Pablo Rubinstein, Gonzalo Ramos e Ignacio Laplagne por su ayuda en distintos momentos del trabajo. Muchas gracias a todos!

Resumen

El problema de *mapping* surge en el contexto de la programación paralela, y fue definido por S. Bokhari en 1981. Una instancia de este problema está dada por un grafo de tareas y un grafo de procesadores, con igual número de nodos. En el primero, los nodos representan las tareas y los ejes indican pares de tareas que se deben comunicar, mientras que el grafo de procesadores representa la arquitectura paralela (los nodos indican los procesadores y los ejes unen pares de procesadores conectados). El objetivo del problema es asignar cada tarea a un solo procesador (cada procesador puede recibir sólo una tarea), de modo tal de maximizar los pares de tareas vecinas que se asignan sobre procesadores conectados.

Se presenta una formulación del problema como un problema de programación lineal entera, en base a la cual se realiza un estudio poliedral, con el objetivo principal de implementar un algoritmo *branch&cut* para su resolución. Se encuentra un conjunto de desigualdades válidas para el poliedro asociado, que constituyen la base fundamental de un algoritmo de este tipo. Se halla también la dimensión del poliedro, lo cual permite decidir la calidad de las desigualdades válidas, logrando probar que algunas de ellas definen facetas.

Se describe la implementación del algoritmo *branch&cut*, realizada con el entorno ABACUS, y se presentan los resultados obtenidos. La implementación de una heurística primal basada en la técnica *chained local optimization* contribuye a reducir sensiblemente los tiempos de ejecución.

Abstract

The mapping problem arises in parallel programming environments, and was first defined by S. Bokhari in 1981. An instance of this problem is given by a task interaction graph and a processors graph, with the same number of vertices. The vertices of the first graph represent the tasks, and its edges join pairs of tasks with communication demands. The second graph represents the parallel arquitecture (its vertices denote processors, which are joined by an edge when they are directly connected). The objective of the problem is to assign each task to one processor (each processor can receive only one task) such that the number of adjacent tasks assigned to connected processors is maximum.

An integer programming formulation for this problem is presented. A polyhedral investigation is carried out, aiming at the implementation of a branch&cut algorithm for this problem. A number of valid inequalities are presented, which constitute the cornerstone of these algorithms. The dimension of the associated politope is found, which allows to determine the strength of these inequalities, proving in some cases that they are facet-defining.

The implementation, performed with the framework ABACUS, *is described and its computational results are presented. The implementation of a primal heuristic, based on chained local optimization metaheuristic, contributes to reduce execution times.*

Contenidos

"El universo había sido creado ex profeso, manifestaba el círculo. En cualquier galaxia que nos encontremos, tomamos la circunferencia de un círculo, la dividimos por su diámetro y descubrimos un milagro: otro círculo que se remonta kilómetros y kilómetros después de la coma decimal. Más adentro, habría mensajes más complejos. Ya no importa qué aspecto tenemos, de qué estamos hechos ni de dónde provenimos. En tanto y en cuanto habitemos en este universo y poseamos un mínimo talento para la matemática, tarde o temprano lo descubriremos porque está aquí, en el interior de todas las cosas. No es necesario salir de nuestro planeta para hallarlo. En la textura del espacio y en la naturaleza de la materia, al igual que en una gran obra de arte, siempre figura, en letras pequeñas, la firma del artista."

– Carl Sagan

Capítulo 1

Introducción

"Pero dando otro paso más atrás, podría decirse que quizá el Universo no ha sido creado sino descubierto en una Selva de Universos Posibles, selva difícil, oscura, sublime, en que sólo un Dios puede aventurarse."

–Ernesto Sabato

Desde sus comienzos durante la Segunda Guerra Mundial, la disciplina conocida como *investigación operativa* ha experimentado un rápido desarrollo, acompañado de resonantes aplicaciones en los más diversos ámbitos. Esta área de la matemática aplicada estudia problemas relativos a la conducción y organización de recursos y actividades en una organización, y la lista de aplicaciones exitosas de sus resultados es virtualmente infinita. Sus contribuciones no sólo incluyen la mejora de la productividad de innumerables empresas comerciales, sino también importantes progresos en la actividad policial, sanitaria y militar (cfr. [Hil/95]).

En el marco de esta disciplina se encuentra la *optimización combinatoria*, que estudia un conjunto más pequeño de problemas, y que constituye uno de los pilares sobre los cuales se asientan los grandes resultados y aplicaciones de la investigación operativa. En un problema de optimización se busca el máximo o mínimo de una cierta función, definida sobre algún dominio. Las teorías clásicas de optimización tratan el caso en el cual el dominio es continuo, mientras que en la optimización combinatoria el dominio es esencialmente discreto (cfr. [Jun/94]).

Problema de optimización combinatoria. *Dado un conjunto finito S (conjunto de soluciones factibles) y una función $f : S \to \Re$ (función objetivo), encontrar un elemento $s^* \in S$ con*

$$f(s^*) = max\ \{f(s) : s \in S\}.$$

Claramente, cuando el dominio es finito, se pueden generar todos sus elementos y elegir el mejor, pero el tamaño de los dominios involucrados habitualmente hace que esta solución resulte computacionalmente imposible. Este problema de optimización es prácticamente insoluble si no se tiene una caracterización de los elementos del dominio S y una forma algorítimica

para evaluar la función objetivo sobre los elementos de S. Normalmente, esta caracterización es sencilla, y en muchos casos corresponde a subgrafos particulares de un grafo dado, disposiciones ordenadas de un conjunto de elementos o biyecciones entre conjuntos finitos. Sin embargo, a pesar de contar con una forma sencilla de identificar los elementos del dominio (y aunque la función objetivo sea trivial), muchos de los problemas que surgen en este contexto pertenecen al grupo de problemas *NP-Hard*, con lo cual es imposible su resolución en forma eficiente, salvo que P=NP.

En estos casos, en los cuales el óptimo es difícil de hallar, se han diseñado *algoritmos heurísticos*, que producen soluciones subóptimas pero cercanas a este valor. En muchos casos es posible acotar la distancia entre la solución arrojada por estos algoritmos y el óptimo del problema, conformando una garantía de la calidad de la solución. También son muy importantes las *heurísticas duales*, que proporcionan (por ejemplo, para un problema de maximización) cotas superiores del óptimo, y que proveen medios para evaluar la calidad de la solución obtenida por una heurística.

A pesar de la intratabilidad de estos problemas, se han propuesto métodos de resolución exacta que han logrado buenos resultados en algunos casos. Entre ellos, los métodos con mejor performance son los llamados *algoritmos branch&cut*, que se basan en programación lineal y programación lineal entera. El presente trabajo describe la construcción e implementación de un algoritmo de este tipo para el problema de *mapping* simple, que surge en el contexto de la programación paralela y constituye uno de los modelos básicos de optimización en esta disciplina. Con este objetivo, se detallan progresivamente todos los aspectos de su construcción, poniendo especial énfasis en el estudio poliedral realizado.

El resto de la presente introducción describe con mayor detalle los algoritmos de planos de corte y las ideas generales de los algoritmos *branch&cut*. En el capítulo 2, se describe el problema de *mapping* simple, reseñando brevemente el contexto en el cual se inscribe. Se mencionan sus aplicaciones y se enumeran los trabajos anteriores sobre este problema. Como sucede con muchos problemas de optimización combinatoria, su enunciado es sencillo pero su resolución exacta es computacionalmente muy difícil. En este sentido, se muestra que el problema pertenece a la clase de problemas *NP-Hard*, y que sigue siéndolo cuando se restringe a los casos particulares que aparecen con frecuencia en la práctica.

Cuando el problema a resolver es *NP-Hard*, como en este caso, uno de los caminos a seguir es la construcción de métodos heurísticos, que obtengan soluciones factibles razonablemente buenas en tiempos cortos. Este tipo de métodos es útil en situaciones en las cuales los tiempos de ejecución son muy importantes, y es suficiente obtener una buena solución aunque no sea óptima. Por otra parte, estos métodos tienen especial interés como heurísticas primales para un algoritmo *branch&cut*, y efectivamente su incorporación reduce notablemente los tiempos de ejecución. En el capítulo 3 se describe la implementación de dos algoritmos basados en técnicas metaheurísticas para este problema. El primero de ellos es una implementación directa del método de rebotes simulados, que se detalla en este mismo capítulo, mientras que el segundo combina las ideas de *simulated annealing* (recocido simulado) con un procedimiento de búsqueda local. Se resumen los resultados de las pruebas computacionales realizadas con estos algoritmos.

El capítulo 4 constituye un "rodeo" en la descripción del algoritmo *branch&cut*. Este capítulo describe un problema dual del problema de *mapping*, que surgió como resultado de la búsqueda de cotas superiores del óptimo. A pesar de no tener aplicación práctica porque las instancias del dual son demasiado grandes, tiene interés teórico por su naturaleza combinatoria (son muy pocos los problemas con duales esencialmente combinatorios). Se realizaron experimentos computacionales con este dual, y se presentan sus resultados en este capítulo.

El siguiente capítulo continúa la descripción del desarrollo del algoritmo *branch&cut*, y describe el núcleo del método implementado. El estudio del poliedro asociado con el problema es fundamental para la construcción de estos algoritmos, y está orientado a encontrar familias de desigualdades válidas para las soluciones enteras. En este capítulo se presentan dos modelos de programación lineal entera equivalentes para el problema, y se encuentra la dimensión de su cápsula convexa. Se describen luego las familias de desigualdades válidas halladas, indicando los casos en los cuales se definen facetas del poliedro.

Las desigualdades válidas particulares para el problema son la piedra fundamental del algoritmo *branch&cut*, que se describe en el capítulo 6. Se enumeran las familias de desigualdades que se incorporaron al algoritmo y se detallan sus procedimientos de separación asociados. Por otra parte, se mencionan las características principales del método (uso de heurísticas, estrategias particulares, fijado de variables, etc.).

Se implementó el algoritmo utilizando el entorno ABACUS (cfr. [Jun/97c], [Jun/97d], [Thi/95], [Thi/96]), junto con el paquete de optimización CPLEX para problemas de programación lineal (cfr. [Cpl/94]). El capítulo 7 presenta los resultados de las pruebas realizadas con esta implementación. Estas pruebas se orientaron a conocer el comportamiento del algoritmo, y a evaluar su performance sobre distintos grupos de instancias. Se realizaron experimentos para (a) determinar empíricamente la "fuerza" de las desigualdades, (b) ajustar los parámetros del método y (c) comparar su rendimiento con el paquete CPLEX de optimización, hasta ahora el único método exacto para la resolución de este problema. Por último, el capítulo 8 resume las conclusiones del trabajo realizado.

Un anexo completa la descripción del desarrollo del algoritmo, en el que se describen los detalles particulares de la implementación, incluyendo la jerarquía de clases, las estructuras de datos, y los métodos específicos utilizados. Este anexo puede resultar de interés para nuevas implementaciones sobre este mismo entorno, puesto que describe con detalle el trabajo realizado.

1.1 Programación lineal y programación lineal entera

En esta y en las siguientes secciones se describen brevemente las ideas fundamentales sobre las que se basan los algoritmos *branch&cut* y el contexto en el cual se aplican. Los problemas de optimización combinatoria están relacionados con dos modelos de optimización: la programación lineal y la programación lineal entera. La programación lineal es uno de los modelos básicos en esta área.

Problema de programación lineal. *Dada una matriz* $A \in \Re^{m \times n}$ *y vectores* $b \in \Re^m$, $c \in \Re^n$, *encontrar un vector* $x^* \in \Re^n$ *con* $Ax^* \leq b$ *y*

$$c^t x^* = max\ \{c^t x : Ax \leq b\}.$$

Los problemas lineales han sido estudiados intensivamente, y existen varios métodos para su resolución, cuyo mayor exponente es el método *simplex*, descubierto por George Dantzig en 1947. A pesar de tener un peor caso exponencial, su tiempo esperado es polinomial y ha demostrado ser extremadamente eficiente en la práctica, utilizándose en forma rutinaria para resolver instancias de gran tamaño. Posteriormente, se hallaron otros métodos con peor caso polinomial, probando que este problema pertenece a la clase P.

Sin embargo, muy pocos problemas que aparecen en optimización combinatoria pueden modelarse como problemas continuos, y en muchos casos debe agregarse como restricción adicional que algunas o todas las variables tomen valores enteros. En este caso, tenemos el problema general de programación lineal entera mixta:

Problema de programación lineal entera mixta. *Dada una matriz* $A \in \Re^{m \times n}$, *vectores* $b \in \Re^m$, $c \in \Re^n$, *y un subconjunto* $I \subseteq \{1, ..., n\}$, *encontrar un vector* $x^* \in \Re^n$ *con* $Ax^* \leq b$ *y* x_i *entera para* $i \in I$, *donde*

$$c^t x^* = max\ \{c^t x : Ax \leq b,\ x_i \text{ entera para todo } i \in I\}.$$

Este problema se denomina **problema de programación lineal entera** si todas las variables deben ser enteras (es decir, si $I = \{1, ..., n\}$). En muchos casos, las variables del conjunto I deben estar entre 0 y 1, en cuyo caso el problema se convierte en el **problema (mixto) de optimización 0-1**. Puede probarse que todos estos problemas son *NP-Hard*.

Muchos problemas de optimización combinatoria pueden expresarse como problemas de programación lineal entera, considerando los vectores característicos de las soluciones factibles. Es decir, para cada solución factible $s \in S$, se define un vector $x^s \in \Re^n$ que la representa. Por ejemplo, si las soluciones factibles son subconjuntos de nodos de un grafo $G = (V, E)$ con $V = \{1, ..., n\}$, puede representarse una solución mediante un vector de n coordenadas, de modo tal que

$$x_i^s = \begin{cases} 1 & \text{si el nodo } i \text{ pertenece a la solución } s \\ 0 & \text{en caso contrario.} \end{cases}$$

Si las soluciones factibles deben cumplir alguna propiedad estructural (por ejemplo, conformar un conjunto independiente), se deben agregar restricciones lineales que expresen esta condición. En este ejemplo, se pueden considerar las siguientes restricciones:

$$x_i + x_j \leq 1, \qquad \forall ij \in E$$

Estas condiciones indican que no pueden estar presentes en la solución los dos extremos de un eje. Como se tiene una restricción por cada eje, el conjunto de nodos representado será un conjunto independiente.

De esta forma, se pueden modelar los problemas de optimización combinatoria como problemas de programación lineal entera, e intentar su resolución en este contexto. Los

recientes avances logrados en la resolución de problemas de optimización combinatoria se han logrado al trasladar estos problemas a problemas de optimización lineal entera (como se mostró anteriormente), para utilizar luego técnicas de programación lineal.

1.2 Algoritmos de planos de corte

Los algoritmos de planos de corte constituyen uno de los métodos para intentar la resolución de problemas de programación lineal entera (cfr. [Aar/95b], [Jun/94]). Como el problema general de programación lineal entera es *NP-Hard*, no se conocen algoritmos polinomiales para su resolución. Se presenta a continuación una introducción a los conceptos básicos de estos algoritmos, que aunque tengan un peor caso exponencial y habitualmente no resulten muy eficientes en la práctica, proporcionan uno de los conceptos de mayor utilidad para los algoritmos *branch&cut*.

Consideremos un problema de optimización combinatoria representado por medio de un problema lineal entero, y llamemos S al conjunto de puntos factibles:

$$S = \{x \in \Re^n : Ax \leq b,\ x_i \text{ entera para } i \in I\}.$$

Podemos asociar a este problema el poliedro $P_S := conv(S)$, formado por la cápsula convexa de los puntos de S. Los extremos de este poliedro serán soluciones factibles del problema, con lo cual podemos resolver el problema de optimización combinatoria resolviendo el problema lineal $\max\{c^t x : x \in P_S\}$, dado que el óptimo de este último se encuentra sobre alguno de los extremos de P_S. Sin embargo, no se conoce ningún algoritmo eficiente para calcular la cápsula convexa P_S de un conjunto de puntos S *definido en forma implícita*, tal como se realiza en un modelo de programación lineal entera.

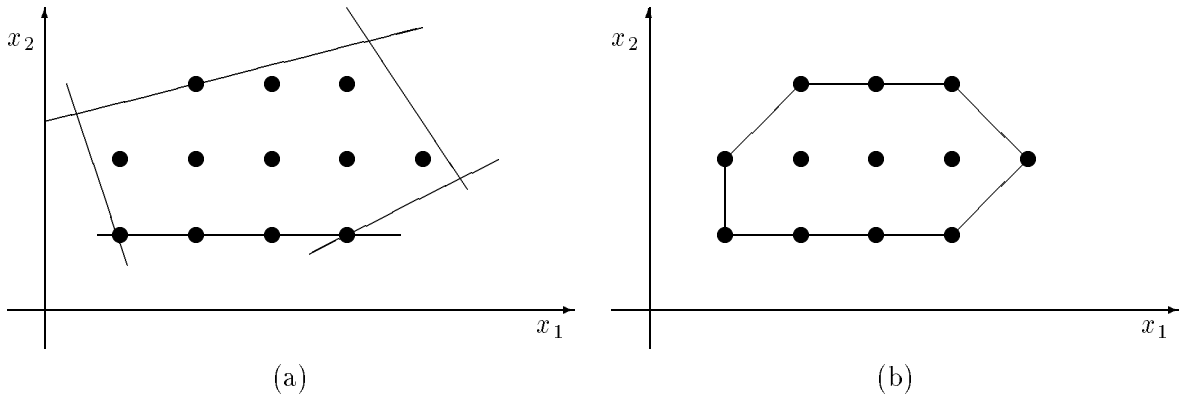


Figura 1.1: (a) Relajación lineal. (b) Cápsula convexa.

Ejemplo. Las restricciones definen un poliedro en $\Re^2$, con una cantidad infinita de puntos (fig. 1.1a), pero sólo nos interesan los puntos enteros que se encuentren dentro de esta región. Si pudiéramos conocer la cápsula convexa de estos puntos (fig. 1.1b), se resolvería el problema en tiempo polinomial. Las restricciones que definen la cápsula convexa se denominan *facetas* del poliedro.

Si olvidamos la condición que dice que x_i ($i \in I$) debe ser entera, tenemos el problema lineal $\max\{c^t x : Ax \leq b\}$, que puede resolverse en tiempo polinomial. El conjunto $S' = \{x \in \Re^n : Ax \leq b\}$ se demonima la *relajación lineal* del problema, y las soluciones factibles del problema inicial son aquellos puntos $x \in S'$ con x_i entera para $i \in I$.

Si resolvemos este nuevo problema lineal, tendremos su óptimo x^*, que puede no ser una solución entera (si lo fuera, puede probarse que también es óptimo del problema entero). En este caso, podríamos hallar una desigualdad $\alpha x \leq \beta$ *que sea cumplida por todos los puntos enteros*, pero tal que $\alpha x^* > \beta$. Si agregamos esta desigualdad al conjunto de restricciones, tendremos un conjunto más pequeño, pero que incluye todas las soluciones enteras y ha "dejado afuera" al punto no entero x^*. Podemos resolver este nuevo problema lineal (encontrando un óptimo distinto) y repitiendo el proceso, hasta hallar una solución entera. El siguiente pseudocódigo describe este método básico para la resolución de problemas enteros.

---

1. Resolver el problema lineal $\max\{c^t x : Ax \leq b\}$, para obtener una solución x^*.
2. Si x_i^* es entera (para $i \in I$), entonces x^* es óptimo del problema lineal entero original. El algoritmo termina.
3. Si alguna coordenada x_i^*, con $i \in I$, no es entera, hallar una desigualdad válida (es decir, una desigualdad que sea cumplida por todas las soluciones enteras) $\alpha x \leq \beta$, de modo tal que $\alpha x^* > \beta$.
4. Agregar la desigualdad al conjunto de restricciones, y volver al paso 1.

Ejemplo. Supongamos que hemos resuelto la relajación lineal del problema, obteniendo el óptimo que muestra la flecha (fig. 1.2a). Como este punto no es entero, generamos un corte que deja afuera a este punto (fig. 1.2b). La figura 1.2c muestra el nuevo poliedro obtenido al agregar este corte a la formulación, cuya relajación se vuelve a resolver, obteniendo el óptimo indicado por la flecha. Nuevamente, este punto no es entero, con lo cual se genera un nuevo corte, indicado por la flecha (fig. 1.2d), y se vuelve a resolver la relajación. El óptimo hallado (fig. 1.2d) es un punto entero, con lo cual es también óptimo del problema original.

El problema principal que presenta este esquema algorítmico es la generación de una desigualdad válida (cumplida por todos los puntos enteros) violada en el óptimo x^* actual. Este problema se denomina **problema de separación**, y puede buscar desigualdades de tres tipos distintos:

1. Desigualdades **sin estructura**, basadas sólo en las variables que deben ser enteras o binarias. Pueden no ser muy fuertes.
2. Desigualdades **de estructura relajada**, que se derivan de relajaciones del problema (por ejemplo, considerando una sola fila del conjunto de restricciones). Por este motivo, sólo pueden separar los puntos fraccionarios que no se encuentren dentro de la cápsula convexa de estas relajaciones. Las desigualdades para el problema de la mochila aplicadas a una restricción particular del problema son un ejemplo de este grupo (cfr., por ejemplo, [Nem/88] seccin II.2.2).

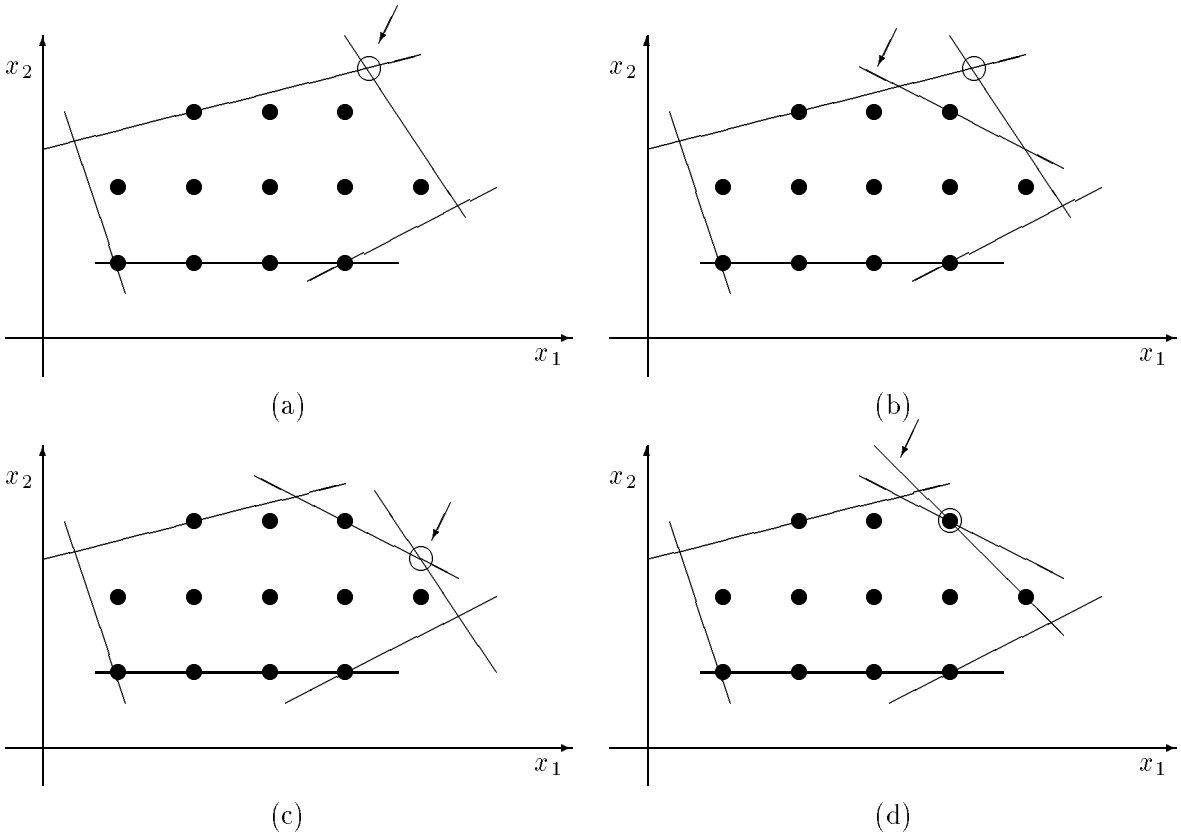


Figura 1.2: Ejemplo de ejecución de un algoritmo de planos de corte

3. Desigualdades **específicas del problema**. Estas desigualdades se derivan típicamente de toda la estructura del problema, o de una parte sustancial. Pueden ser muy fuertes, aunque su aplicación está limitada al problema particular del cual se obtienen.

El camino que ha producido mejores resultados es el estudio de los poliedros asociados con cada problema particular, para encontrar desigualdades válidas específicas para cada problema (cfr. [Fer/93], [Gro/95], [Aar/95a]). El procedimiento habitual consiste en hallar familias de desigualdades válidas, tan fuertes como sea posible. Además de contar con desigualdades válidas, es necesario definir *procedimientos de separación*, que reciban el óptimo x^* actual y generen aquellas desigualdades de la familia que sean violadas en este punto, posiblemente en forma heurística.

Este método de resolución no es eficiente en la práctica, pero se logran los mejores resultados cuando se combina con técnicas de enumeración (*branch&bound*), para conformar un algoritmo conocido como *branch&cut*, que se describe a continuación.

1.3 Algoritmos *branch&cut*

Como se mencionó, los métodos de planos de corte no son eficientes e incluso pueden no hallar el valor óptimo. Dependiendo de la estrategia de generación de cortes, el método puede quedarse "trabado" en una solución no factible, sin poder detectar nuevos cortes que eliminen este punto. Se puede evitar esta situación utilizando otra técnica algorítimica básica para esta clase de problemas, dada por los métodos *branch&bound* (enumeración implícita). Un método *branch&bound* constituye un esquema de *divide and conquer*, que intenta resolver el problema original dividiendolo en subproblemas más pequeños, para los cuales se computan cotas inferiores y superiores (cfr. [Joh/98], [Jun/94]). Nuevamente, este enfoque no produce buenos resultads en la práctica, pero proporciona uno de los conceptos básicos de los algoritmos *branch&cut*.

La resolución recursiva de los subproblemas origina una enumeración exhaustiva de todas las soluciones factibles, y es computacionalmente imposible. Al mantener en forma simultánea los subproblemas aún no resueltos, y calcular cotas inferior y superior del óptimo, los algoritmos *branch&bound* pueden reducir el número de pasos de la enumeración, obteniendo tiempos de ejecución mucho menores.

Específicamente, el método mantiene una lista de subproblemas abiertos (formada inicialmente por el problema original) de la cual se selecciona un elemento en cada paso. Llamemos S al conjunto de soluciones factibles de este subproblema. Resolvemos alguna relajación del subproblema (por ejemplo, la relajación lineal), para obtener una cota superior del óptimo. Si la solución es factible entera, se ha encontrado el óptimo del subproblema, que se elimina de la lista de nodos activos. Si no, recurrimos a la enumeración. Dividimos el problema en dos (o más) subproblemas, que se incorporan a una lista de subproblemas abiertos. En cada paso, se actualizan las cotas inferior (mejor solución hallada hasta el momento) y superior. Veamos con más detalle los dos componentes básicos del algoritmo:

Branching. La subdivisión en subproblemas debe particionar el conjunto de soluciones, y debe ser eficiente (asumimos que los subproblemas generados son del mismo tipo que el problema original). Este proceso de subdivisiones puede representarse en forma de árbol, llamado *árbol de enumeración*, en el cual los nuevos subproblemas tienen como padre al subproblema del cual provienen. Una técnica habitual de *branching* es generar los nuevos nodos agregando las restricciones $x_i \leq k$ y $x_i \geq k+1$, si x_i^* $(i \in I)$ es fraccionaria y $\lfloor x_i^* \rfloor = k$.

Bounding. Cuando la relajación de un subproblema arroja un valor menor que la cota inferior calculada hasta el momento, entonces ninguna de las soluciones factibles del subproblema podrá ser solución óptima del problema inicial, dado que existe una solución (la que determina la cota inferior actual) con mayor función objetivo. En este caso, puede eliminarse este subproblema de la lista de subproblemas activos (se dice que se "cierra" el nodo del árbol). Por otra parte, si el óptimo de la relajación es solución factible del subproblema, entonces es su óptimo, con lo cual no se subdivide en nuevos subproblemas, y también se elimina de la lista de problemas abiertos.

La característica fundamental de este método es el mantenimiento de cotas inferiores y superiores del óptimo, que se van acercando progresivamente. La cota superior está definida

por las sucesivas relajaciones que se resuelven a lo largo del proceso de enumeración, mientras que las cotas inferiores provienen de las soluciones factibles enteras que aparecen a lo largo del proceso[1]. El algoritmo termina cuando estas cotas coinciden, en cuyo caso la mejor solución factible hallada es óptimo del problema.

El método *branch&cut* es una generalización de *branch&bound* (cfr. [Jun/92], [Jun/94], [Aar/95b]). Si luego de resolver la relajación lineal de un subproblema no se puede cerrar el nodo correspondiente, se busca una desigualdad violada por el óptimo de la relajación. Si se encuentran desigualdades violadas (llamadas habitualmente *cortes*), se agregan a la formulación y se resuelve el problema lineal nuevamente. Si no se encuentra ningún corte, se continúa con el proceso de *branching*. El siguiente pseudocódigo muestra las ideas básicas del método.

---

1. Inicializar el árbol de enumeración.
2. Elegir un nodo abierto del árbol.
3. Resolver la relajación lineal del subproblema asociado al nodo. Sea x^* el óptimo de la relajación ($c^t x^*$ es cota superior del óptimo del subproblema). Actualizar la cota superior (si corresponde).
   (a) Si x^* es solución factible del problema, actualizar la cota inferior (si corresponde). Cerrar el nodo y volver al paso 2.
   (b) Si $c^t x^*$ es menor que la cota inferior, ninguna solución factible del subproblema puede ser óptimo del problema original (*bounding*). Cerrar el nodo y volver al paso 2.
4. Buscar desigualdades violadas por x^*. Si se encuentran, agregarlas a la formulación y volver al paso 3. Si no se encuentran, o si se cumple algún otro criterio (número fijo de iteraciones, control de *tailing off*, etc.), generar dos subproblemas (*branching*) y agregarlos a la lista de problemas abiertos.
5. Si las cotas inferior y superior coinciden, entonces la mejor solución factible hasta el momento es el óptimo del problema. En caso contrario, volver al paso 2.

El método *branch&cut* combina y generaliza los dos métodos de resolución (algoritmos de planos de corte puro y *branch&bound*). Se han desarrollado algoritmos basados en este método para un gran número de problemas de optimización combinatoria, logrando resolver instancias de gran tamaño en algunos casos (cfr. [Asc/97], [Fer/93]], [Gro/92], [Jun/92], [Mag/97]). Hasta la fecha, este enfoque ha resultado el más efectivo para la resolución exacta de esta clase de problemas.

[1] Pueden utilizarse heurísticas que recorran el espacio de soluciones y proporcionen cotas inferiores, previamente o durante la ejecución del algoritmo (cfr. [Jun/92]).

Capítulo 2

El problema de mapping simple

"Si el hombre, con su intelecto creado y con las limitaciones de la propia subjetividad, pudiese superar la distancia que separa la creación del Creador (...), sólo entonces sus preguntas estarían fundadas."

–Juan Pablo II

Muchas aplicaciones científicas requieren una potencia computacional que se encuentra más allá de las posibilidades de las computadoras secuenciales de la actualidad, y que sólo puede ser alcanzada con el uso de computadoras paralelas. Se han propuesto muchas arquitecturas de computadoras paralelas, en las cuales los elementos de procesamiento se conectan por medio de una red de comunicación. Estas arquitecturas pueden clasificarse en dos grupos. El primer grupo de arquitecturas, llamadas *arquitecturas dedicadas*, intenta obtener una buena performance para una clase particular de trabajos. Las arquitecturas del segundo grupo, llamadas *de propósito general*, están diseñadas de modo tal de proveer una buena performance promedio para una gran cantidad de trabajos, y es en este grupo donde la planificación tiene importancia decisiva.

En efecto, uno de los problemas cruciales que se deben resolver en este contexto es el desarrollo de estrategias para sacar el mayor provecho de la potencia de cálculo de los equipos. Una mala estrategia puede producir *overheads* innecesarios en la comunicación y sincronización de los procesadores, tiempos muertos debidos al uso compartido de recursos, o esperas superfluas debidas a una inadecuada consideración de restricciones de precedencia. Cobra relevancia en este ámbito el modo en el cual se utilizan los recursos computacionales disponibles y los métodos aplicados para tomar estas decisiones.

Puede expresarse este problema en términos de un conjunto de tareas que se deben ejecutar sobre los procesadores de la arquitectura, respetando un conjunto de restricciones (de precedencia, de comunicación, etc.). Cuando los procesos tienen una duración limitada y sus dependencias lógicas se tienen en cuenta, tenemos un problema denominado de *scheduling*. Cuando se asume que los procesos coexisten simultáneamente durante toda la ejecución, estamos frente a un problema de *mapping*. En este último caso, se busca reducir los costos

de comunicación (manteniendo los procesos con mucha comunicación en procesadores conectados), balanceando la carga computacional entre todos los procesadores. Una asignación (*mapping*) se denomina *estática* si se computa antes de la ejecución del programa, y no se modifica durante la ejecución (cfr. [Pel/99]).

En muchos casos, puede modelarse el problema por medio de grafos que representan las relaciones lógicas entre las tareas y la arquitectura paralela. Muchos modelos de *scheduling* consideran al conjunto de tareas como un grafo dirigido, en el cual los vértices representan las tareas, y los ejes representan las dependencias de ejecución. Es decir, si un eje lleva de una tarea a otra, entonces la primera debe completarse antes que la última pueda comenzar su ejecución. Pueden asociarse pesos con los nodos, indicando el tiempo o cantidad de instrucciones (cfr. [Por/92]) de cada tarea, y con los ejes, que representan el tiempo de comunicación entre sus extremos (cfr. [Bar/98]). El objetivo de este modelo puede ser, por ejemplo, minimizar el tiempo de finalización de todos los procesos.

Por otra parte, en problemas de *mapping*, se tiene habitualmente un grafo no dirigido, cuyos vértices nuevamente representan a las tareas y sus ejes denotan la necesidad de comunicación entre los procesos (dos tareas que se deben comunicar comparten un eje). En muchos de estos modelos no se representan explícitamente las dependencias temporales, con lo cual todas las tareas se consideran simultáneas e independientes. El objetivo de este problema puede ser minimizar los costos de comunicación o balancear la carga entre los procesadores. El problema de *mapping* simple pertenece a este modelo de problemas.

2.1 El problema de *mapping* simple

El *problema de mapping simple* se inscribe dentro del *modelo de interacción de tareas*, y considera tareas simultáneas e independientes, pero que se deben comunicar entre sí.

Problema de *mapping* simple. *Sean G y H dos grafos. Los nodos del grafo G representan las tareas que se deben ejecutar, y los ejes de $E(G)$ indican los pares de tareas que se deben comunicar. Por su parte, el grafo H representa la arquitectura paralela, siendo $V(H)$ los procesadores, y $E(H)$ los pares de procesadores que se encuentran unidos por una conexión directa. Se asume que $|V(G)| = |V(H)|$ (es decir, existe el mismo número de tareas y procesadores). El problema consiste en asignar las tareas a los procesadores (una tarea a cada procesador), de modo tal de maximizar la cantidad de pares de tareas adyacentes que se asignan a procesadores conectados. Es decir, se busca maximizar la cantidad de ejes de $E(G)$ cuyos extremos se asignan a procesadores adyacentes en $E(H)$.*

Debe notarse que cada tarea se asigna a un sólo procesador, y que cada procesador puede recibir una sola tarea. Una solución factible de este problema está definida por una biyección $f : V(G) \to V(H)$, que asigna un procesador a cada tarea, formando una correspondencia biunívoca entre los nodos del grafo de tareas y los nodos del grafo de procesadores. Esta biyección se llamará en adelante "asignación de tareas a procesadores", o más brevemente, "asignación".

La cantidad de ejes de G que se asignan sobre ejes de H se denomina la *cardinalidad* de

la asignación. Las tareas adyacentes que sean asignadas a procesadores conectados se podrán comunicar por medio de la conexión entre sus procesadores respectivos, y esta comunicación será muy rápida. Si estas tareas se asignaran a procesadores que no están conectados, entonces deberán comunicarse por medio del bus global, con lo cual la comunicación entre ellas será más lenta. Por este motivo, se busca maximizar la cardinalidad de la asignación.

Ejemplo. La figura 2.1 muestra una instancia del problema. La figura 2.1a muestra el grafo de tareas, que se enumeran entre 0 y 5, y la figura 2.1b muestra el grafo de procesadores, ambos de 6 nodos. Si la tarea 0 se asigna al procesador 0, la tarea 1 al procesador 1, etc., la asignación obtenida tiene cardinalidad 5. La figura 2.1c muestra los ejes del grafo de tareas que caen sobre ejes del grafo de procesadores.

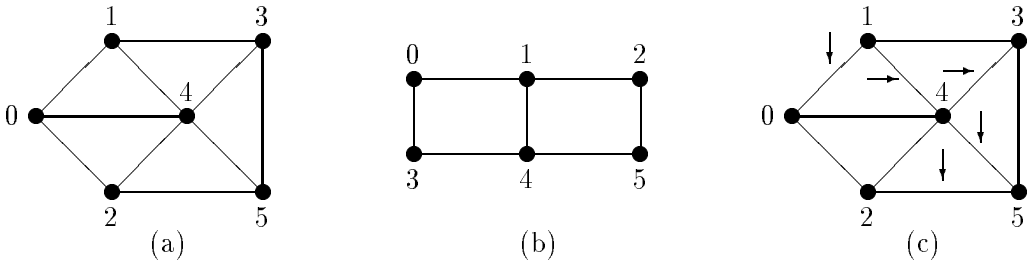


Figura 2.1: Ejemplo de una instancia de 6 nodos.

Cuando la cantidad de tareas es menor que la cantidad de procesadores, pueden agregarse tareas aisladas (que no se comunican con ninguna otra tarea) hasta igualar el número de procesadores. Por otra parte, Cuando $|E(G)| = |E(H)|$ (es decir, G y H tienen la misma cantidad de ejes), entonces G y H son isomorfos si y sólo si existe una asignación de cardinalidad $|E(G)|$. Con esto, el problema de isomorfismo de grafos es un caso particular del problema de *mapping*, lo cual constituye una segunda motivación para su estudio. Cuando los grafos no son isomorfos, el óptimo del problema de *mapping* indica "la mejor asignación", es decir, la que respeta el mayor número de ejes.

Ejemplo. Si tenemos el grafo de tareas del ejemplo anterior (fig. 2.1a), pero debe asignarse sobre una arquitectura de 8 nodos, como en la figura 2.2b, entonces podemos agregar dos nuevas tareas aisladas (las tareas 6 y 7), como muestra la figura 2.2a.

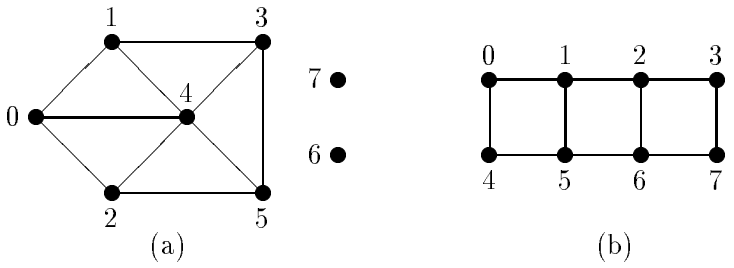


Figura 2.2: Una instancia de 8 nodos, con dos tareas aisladas.

2.1.1 Estudios previos del problema de *mapping*

El problema de *mapping* simple fue definido por Bokhari (cfr. [Bok/81]), y su función objetivo se basa en la cardinalidad de las asignaciones. Bokhari propone una heurística para la máquina FEM (*Finite Element Machine*), que tiene una arquitectura en grilla de 8 vecinos por nodo. Su heurística se basa en intercambios de a pares, alternando con saltos probabilísticos, y se asemeja al método *Chained Local Optimization*, descripto en el capítulo 3. Desde ese momento, se han propuesto distintas variaciones del problema (que utilizan otras medidas para evaluar la calidad de las asignaciones) y se han presentado diferentes algoritmos heurísticos, en su mayoría basados en conceptos sobre grafos.

El modelo de problemas de *mapping* es aplicable a la mayoría de las implementaciones para resolver ecuaciones diferenciales parciales. En este sentido, Sadayappan y Ercal [Sad/87] proponen una función de costo muy similar a la del problema de *mapping* simple para esta clase de aplicaciones, y presentan un algoritmo heurístico basado en particionamiento de grafos, para arquitecturas de procesadores en grilla.

Otra heurística basada en conceptos sobre grafos fue propuesta por Ercal et al [Erc/90], pero se aplica a una variación del problema en la que se utiliza una función objetivo distinta. En esta variación, el grafo de tareas tiene pesos en los ejes, que indican el costo de las comunicaciones, y la función objetivo es una estimación del costo total de comunicación (se trata de una función objetivo heurística). Realizan, además, una comparación entre este método y una implementación de la técnica metaheurística *simulated annealing*.

Otra implementación desde el punto de vista de las técnicas metaheurísticas fue realizada por Chockalingam y Arunkumar [Cho/95], que presentan un método basado en algoritmos genéticos. La función objetivo que consideran incluye un término de penalización que intenta balancear la carga de la comunicación. Por otra parte, Lee y Aggarwal [Lee/87] definen un conjunto de funciones objetivo posibles, con la intención de caracterizar con mayor precisión el *overhead* de la comunicación. Proponen un nuevo modelo con su función objetivo, que es aplicable al procesamiento paralelo de imágenes, y presentan un algoritmo heurístico basado en una asignación inicial seguida de intercambios de a pares.

En resumen, el problema tiene un rango amplio de aplicación (cfr. [Cho/95]) y resulta un modelo muy aplicable para muchos problemas que surgen en el contexto de la programación paralela. De acuerdo con los artículos citados, todos los acercamientos al problema se han realizado desde el ámbito de la programación paralela, proponiendo heurísticas que en muchos casos son válidas sólo para arquitecturas particulares. En el presente trabajo, se aplican técnicas y modelos propios de la investigación operativa y de la programación lineal entera, para implementar un algoritmo exacto de tipo *branch&cut*, basado en un estudio poliedral del problema. No tenemos conocimiento de que se haya intentado este enfoque previamente. Por estos motivos, se trata de un estudio inicial, y los resultados que se obtengan estarán en concordancia con esta situación.

El primer paso que se realiza en el estudio del problema es la determinación de su complejidad computacional. Todos los artículos citados mencionan que el problema general (en todas sus variaciones) pertenece a la clase de problemas NP-Hard. Sin embargo, el proble-

ma general admite cualquier tipo de arquitectura, y puede suceder que deje de pertenecer a esta categoría cuando se restringe sobre arquitecturas particulares. Puede probarse que el problema sigue siendo NP-Hard aún cuando se consideran arquitecturas puntuales (se prueba este resultado para arquitecturas en grilla de 4, 6 y 8 vecinos por nodo). En la siguiente sección se presenta una demostración de la dificultad del problema general, para probar luego la intratabilidad de los problemas restringidos sobre arquitecturas particulares.

2.2 Complejidad del problema

Se analiza a continuación la complejidad del problema de *mapping*, para el cual pudimos probar que se encuentra dentro de la clase de problemas NP-completos[1]. Este resultado torna improbable la construcción de algoritmos polinomiales para su resolución, dado que no se conocen algoritmos de este tipo para los problemas NP-completos, y muchas personas creen que esto sucede porque efectivamente no existen tales algoritmos.

Teorema. *El problema de decisión asociado al problema de mapping es NP-completo.*

Demostración. El problema de decisión asociado recibe dos grafos (de tareas y de procesadores) con la misma cantidad de nodos y un entero k, y consiste en decidir si existe una asignación de cardinalidad k o mayor. No es difícil comprobar que se encuentra en NP, dado que un programa podría generar una asignación en forma no determinística y verificar si su cardinalidad es k o mayor.

Para probar que es NP-completo, se muestra una transformación polinomial desde el problema de clique máxima. Sea I una instancia del problema de clique máxima, dada por un grafo G y un entero k (el problema consiste en decidir si G tiene una clique de k o más nodos). Construimos una instancia $f(I)$ del problema de *mapping* tomando a G como el grafo de tareas y $H = K_k \bigcup \bar{K}_{n-k}$ como el grafo de procesadores, con $n = |V(G)|$. Es decir, H está compuesto por una clique de tamaño k y $n - k$ nodos aislados (fig. 2.3).

Figura 2.3: Ejemplo con $k = 4$.

El grafo G tiene una clique de tamaño k si y sólo si existe una asignación de tareas (nodos de G) a procesadores (nodos de H) de cardinalidad $\frac{k(k-1)}{2}$, dado que una tal asignación cubre

[1] Es preciso mencionar que los artículos que tratan este problema dan por sentado que el problema es NP-completo, pero no nos fue posible hallar una demostración de este hecho.

todos los ejes de H. Además, la transformación es polinomial, con lo cual el problema de decisión asociado al problema de *mapping* es NP-completo.

Corolario. *El problema de mapping es NP-Hard.*

Luego, si P≠NP, no existen algoritmos polinomiales exactos para este problema. La siguiente proposición muestra que es también improbable la existencia de algoritmos polinomiales que garanticen una desviación del óptimo en menos de una constante. Es de interés analizar también la existencia de algoritmos aproximados que garanticen una desviación de un factor respecto del óptimo, pero no nos fue posible hallar una demostración similar en este caso. Si A es un algoritmo, denotamos $A(I)$ el valor que arroja sobre la instancia I del problema, y $OPT(I)$ al óptimo de esa instancia.

Proposición. *Si P≠NP, no existe un algoritmo aproximado A polinomial para el problema de mapping que garantice $|OPT(I) - A(I)| \leq K$ para toda instancia I.*

Demostración. Supongamos que existe un algoritmo A con estas características. Veremos que si este algoritmo existe, lo podemos utilizar para resolver el problema de *mapping* en tiempo polinomial, contradiciendo la suposición P≠NP. Sea I una instancia del problema, definida por los grafos G y H. Formamos una nueva instancia I' tomando $K+1$ copias de G como grafo de tareas y $K+1$ copias de H como grafo de procesadores. Entonces, $OPT(I') = (K+1)OPT(I)$. Por otra parte, una vez ejecutado el algoritmo aproximado, que arroja un resultado de $A(I)$, sabemos que existe una asignación de cardinalidad al menos $\lceil A(I')/(K+1) \rceil$ (puesto que existe una asignación de I' de cardinalidad mayor o igual que $A(I')$ y múltiplo de $K+1$, dada por la repetición de una asignación de I). Además, como

$$|A(I') - OPT(I')| = |A(I') - (K+1)OPT(I)| \leq K,$$

entonces esta asignación tendrá cardinalidad exactamente K. Luego, este procedimiento permite hallar el valor óptimo del problema y es un algoritmo polinomial. La contradicción proviene de suponer que se tiene un algoritmo aproximado, con lo cual no es posible la existencia de un algoritmo con estas características.

2.3 Restricción sobre arquitecturas particulares

Cuando se tiene un problema general y se prueba que es NP-completo, se tiene fuerte evidencia de que no puede resolverse en tiempo polinomial. Sin embargo, existen muchas otras preguntas que este hecho no puede responder. Normalmente, el problema que se analiza es general y está despojado de detalles y estructuras que aparecen en la práctica. Por ejemplo, puede suceder que el problema sea NP-completo, pero que las instancias que se deben resolver tengan particularidades que posibiliten su resolución polinomial. El camino a seguir consiste en analizar los subproblemas que aparecen en la práctica, para ver si siguen perteneciendo a esta clase.

Definición (subproblema). *Sea $\Pi = (D, Y)$ un problema de decisión, con dominio (conjunto de instancias) D y conjunto de instancias afirmativas Y. Decimos que $\Pi' = (D', Y')$ es un subproblema de Π si $D' \subseteq D$, e $Y' = Y \cap D'$.*

Un subproblema responde la misma pregunta que el problema original, pero sobre un conjunto reducido de instancias, que cumplen ciertas restricciones adicionales. En este sentido, las restricciones naturales que surgen para el problema de *mapping* están dadas por las características de las arquitecturas: El problema general se enuncia para cualquier grafo H de procesadores, pero los casos de interés (en la programación paralela) tienen arquitecturas particulares (grillas rectangulares, cubos, hipercubos, etc.). En esta sección se analiza la complejidad de los subproblemas que aparecen al considerar solamente tipos particulares de arquitecturas. Debe mencionarse que no hemos hallado en la literatura ningún análisis de este tipo para el problema de *mapping* simple.

Para realizar este análisis, será necesario considerar el conjunto de problemas NP-completos en sentido fuerte, dado que las demostraciones de esta sección utilizan un problema de esta clase. Dada una instancia I de un problema, definimos dos funciones que describen el "tamaño" de la instancia en términos de su longitud y de los números que aparecen en ella. La función $Length[I]$ indica el número de símbolos utilizados para describir la instancia bajo alguna codificación razonable. Por su parte, la función $Max[I]$ hace corresponder a la instancia la magnitud del número más grande de I. La elección de la función de longitud está dada implícitamente por la elección del esquema de codificación, mientras que la función Max surge de la descripción del problema en términos informales.

Para cualquier problema de decisión Π y cualquier polinomio p, sea Π_p el subproblema de Π obtenido al considerar sólo las instancias con $Max[I] \leq p(Length[I])$. Decimos que el problema Π es *NP-completo en sentido fuerte* si Π se encuentra en NP y existe un polinomio p para el cual Π_p es NP-completo. Si un problema es NP-completo en sentido fuerte, entonces no puede ser resuelto por un algoritmo pseudopolinomial, a menos que P=NP. En este contexto, se realizan con frecuencia transformaciones pseudopolinomiales, en contraste con las transformaciones polinomiales utilizadas para probar resultados de NP-completitud.

Definición (transformación pseudopolinomial). *Sean Π y Π' problemas de decisión con conjuntos de instancias D_Π y $D_{\Pi'}$, conjuntos de instancias afirmativas Y_Π e $Y_{\Pi'}$ y funciones Max, $Length$, Max' y $Length'$, respectivamente. Una transformación pseudopolinomial de Π a Π' es una función $f : D_\Pi \to D_{\Pi'}$ tal que:*

- Para cada $I \in D_\Pi$, $I \in Y_\Pi$ si y sólo si $f(I) \in Y_{\Pi'}$.
- La función f puede computarse en tiempo polinomial en las dos variables $Max[I]$ y $Length[I]$.
- Existe un polinomio q_1 tal que para todo $I \in D_\Pi$,

$$q_1(Length'[f(I)]) \geq Length[I].$$

- Existe un polinomio de dos variables q_2 tal que para todo $I \in D_\Pi$,

$$Max'[f(I)] \leq q_2(Max[I], Length[I]).$$

Las demostraciones de los siguientes lemas pueden hallarse en [Gar/79].

Lema. *Si* Π *es NP-completo en sentido fuerte,* $\Pi' \in NP$ *y existe una transformación pseudopolinomial de* Π *a* Π'*, entonces* Π' *es NP-completo en sentido fuerte.*

Lema. *Si* Π *es NP-completo en sentido fuerte, entonces* Π *es NP-completo.*

Pasamos ahora al teorema central de esta sección, que prueba que la restricción del problema sobre grillas de cuatro vecinos por nodo es NP-completo. Para esto, se reduce el problema 3-PARTITION al problema de *mapping* sobre estas arquitecturas, por medio de una transformación pseudopolinomial. Como el problema 3-PARTITION es NP-completo en sentido fuerte, esto prueba que la restricción sobre estas arquitecturas es NP-completo, con lo cual la existencia de un algoritmo polinomial para su resolución es imposible, a menos que P=NP.

Definición (3-PARTITION). *El problema está definido por un conjunto* A *de* $3m$ *elementos, una cota* $B \in Z^+$*, y un tamaño* $s(a) \in Z^+$ *para cada* $a \in A$*, de modo tal que* $B/4 < s(a) < B/2$ *y tal que* $\sum_{a \in A} s(a) = mB$*. El objetivo del problema es decidir si existe una partición de* A *en* m *conjuntos disjuntos* $A_1, A_2, ..., A_m$*, de modo tal que* $\sum_{a \in A_i} s(a) = B$ *(con lo cual cada* A_i *tiene exactamente 3 elementos).*

Garey y Johnson ([Gar/75]) probaron que 3-PARTITION es NP-completo en sentido fuerte. Se utiliza este resultado para demostrar el siguiente teorema.

Teorema. *La restricción del problema de mapping sobre arquitecturas en grilla con cuatro vecinos por nodo es NP-completo.*

Demostración. Para probar el teorema, se construye una transformación pseudopolinomial de 3-PARTITION al problema de *mapping* sobre estas arquitecturas. Sea I una instancia de 3-PARTITION, con los elementos que solicita la definición de este problema. Sean $a^* = min_{a \in A} s(a)$, y $k = \lceil (2m+1)/a^* \rceil$. La transformación debe construir una instancia $f(I)$ del problema de *mapping*, para lo cual debemos proporcionar los grafos G de tareas y H de procesadores.

El grafo H de procesadores consiste de una grilla rectangular, con cuatro vecinos por nodo, de $2m$ filas y kB columnas. Por su parte, el grafo G de tareas está formado por $3m$ componentes conexas $C(a)$ $(a \in A)$, asociadas con cada elemento de A. Si $a \in A$ tiene peso $s(a)$, su componente asociada $C(a)$ es una grilla de nodos, de 2 filas y $ks(a)$ columnas, de tres vecinos por nodo (excepto los extremos, que tienen dos vecinos). Por ejemplo, la figura 2.4 muestra las componentes asociadas con elementos de pesos 2, 3 y 4, si $k = 1$.

De acuerdo con la construcción, G tiene componentes $C(a)$, cada una con $2ks(a)$ nodos, con lo cual G tiene $\sum_{a \in A} 2ks(a) = 2kmB$ nodos. Por su parte, H es una grilla de $2m$ por

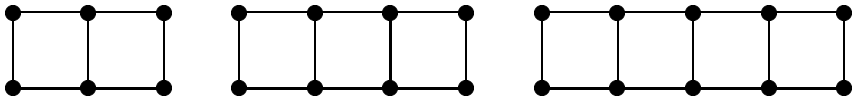


Figura 2.4: Componentes de pesos 3, 4 y 5 ($k = 1$).

kB nodos, con lo cual G y H tienen la misma cantidad de nodos. En el nuevo problema, respondemos "sí" cuando existe una asignación en la cual todos los ejes de G corresponden a ejes de H.

Verificamos ahora las condiciones que debe cumplir f para que sea una transformación pseudopolinomial. En primer lugar, debe suceder que I es una instancia afirmativa de 3-PARTITION si y sólo si $f(I)$ es una instancia afirmativa del problema de *mapping*.

$\Rightarrow$) I es una instancia afirmativa de 3-PARTITION si pueden agruparse sus $3m$ elementos en conjuntos $A_i = \{a_i^1, a_i^2, a_i^3\}$ de tres elementos de modo tal que cada uno de estos conjuntos tenga suma B. Si esto sucede, podemos ubicar las componentes $C(a_i^1)$, $C(a_i^2)$ y $C(a_i^3)$ consecutivamente en forma horizontal, de modo que ocupen exactamente un rectángulo de 2 por kB en H. Repitiendo con los restantes conjuntos, se pueden ubicar todas las componentes de G sobre la grilla, de modo que cada "fila doble" contenga tres componentes ocupando toda la fila (es decir, con suma B). Por lo tanto, el óptimo de $f(I)$ asigna todos los ejes de G a ejes de H.

$\Leftarrow$) Si G puede asignarse sobre H de modo tal que todos sus ejes corresponden a ejes de H, entonces esta asignación mantiene las componentes de G (es decir, cada componente se asigna sobre un subgrafo de H isomorfo a la componente). Como cada una tiene al menos $ka^* \geq (2m + 1)$ columnas, entonces no puede ubicarse en forma vertical. Por lo tanto, las componentes de G se ubican sobre la grilla en forma horizontal y cada fila contiene componentes con un total de kB columnas ocupadas, con lo cual sus elementos asociados suman exactamente B. Por lo tanto, los elementos de A pueden agruparse en conjuntos de tres elementos, cada uno de suma B.

Verificamos ahora las tres condiciones restantes. De acuerdo con la construcción anterior, los grafos G y H tienen $O(mkB)$ nodos cada uno, con lo cual la instancia construida tiene un tamaño polinomial en $Length[I]$ y $Max[I]$, puesto que $m, k, B \leq Max[I]$. Como el agregado de un nodo o un eje a un grafo se realiza en tiempo polinomial, la función f de construcción puede computarse en tiempo polinomial en $Length[I]$ y $Max[I]$ (es decir, en tiempo pseudopolinomial).

Como $Length[I] = O(m \lg B + \lg m)$ y $Length'[f(I)] = O(mkB)$, entonces existe un polinomio q_1 tal que $q_1(Length'[f(I)]) \geq Length[I]$. Por último, $Max'(f[I]) = max\{2m, kB\}$ y $Max(I) = max\{m, B\}$, con lo cual la transformación cumple la cuarta condición. Por lo tanto, de acuerdo con el primer lema, el problema de *mapping* restringido a arquitecturas en grilla con cuatro vecinos por nodo es NP-completo en sentido fuerte. Por el segundo lema, este problema es NP-completo.

Debe notarse que la misma idea de la demostración se aplica a arquitecturas en grilla con seis y ocho vecinos por nodo, considerando distintas componentes $C(a)$, como se muestra en la figura 2.5. Con esto, se tienen los siguientes corolarios de la demostración anterior.

Corolario. *El problema de mapping restringido a arquitecturas en grilla con seis vecinos por nodo es NP-completo.*

Corolario. *El problema de mapping restringido a arquitecturas en grilla con ocho vecinos por nodo es NP-completo.*

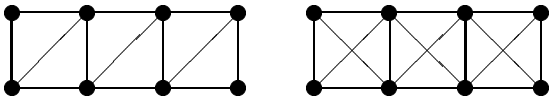


Figura 2.5: Componentes para los problemas con 6 y 8 vecinos por nodo.

2.4 Restricción sobre grafos acíclicos

En la sección anterior se demostró que el problema permanece NP-completo cuando se restringen las arquitecturas a grillas rectangulares de 4, 6 y 8 vecinos por nodo. En esta sección se prueban resultados similares para el problema que surge al restringir el grafo de tareas a grafos acíclicos (bosques). Este caso surge cuando se consideran las tareas agrupadas por conjuntos de modo tal que cada tarea sólo se deba comunicar con las tareas de su mismo conjunto, y dentro de cada conjunto se organiza la comunicación en una estructura de árbol, para llevar al mínimo la cantidad de canales ocupados.

Teorema. *El problema de mapping restringido a grafos de tareas acíclicos es NP-completo.*

Demostración. Se demuestra el teorema construyendo una transformación desde el problema de camino máximo en un grafo[2], que es un problema NP-completo. Una instancia I de este problema está dada por un grafo H y un entero K, y se construye una instancia $f(I)$ del problema de *mapping* restringido del siguiente modo: El grafo de procesadores es H y el grafo de tareas está formado por un camino P de $K+1$ nodos y $n-K-1$ nodos aislados ($n = |H|$). La instancia $f(I)$ será afirmativa si existe una asignación que haga corresponder los K ejes del grafo de tareas a ejes de H (ver fig. 2.6).

No es difícil comprobar que esta transformación es polinomial, con lo cual sólo resta ver que hace corresponder las instancias afirmativas de ambos problemas. Es decir, I es una instancia afirmativa del problema de camino máximo si y sólo si $f(I)$ es una instancia afirmativa del problema de *mapping*.

[2]El problema de camino máximo recibe un grafo y un entero K y consiste en decidir si el grafo tiene un camino de longitud K o mayor.

$\Rightarrow$) Como I es una instancia afirmativa del primer problema, entonces existe un camino de longitud K en H. Luego, pueden asignarse los nodos de P al camino de K (y los restantes nodos aislados en cualquier otro procesador), obteniendose una asignación con K ejes sobre ejes de H. Es decir, $f(I)$ es una instancia afirmativa.

$\Leftarrow$) Si $f(I)$ es una instancia afirmativa, entonces existe una asignación que hace corresponder los K ejes del grafo de tareas a ejes de H. Como estos ejes forman un camino, entonces sus nodos se corresponden con nodos $n_1, ..., n_k + 1 \in V(H)$ tales que $(n_i, n_{i+1}) \in E(H)$. Por lo tanto, existe un camino de longitud K en H, e I es una instancia afirmativa del problema de camino máximo.

Por lo tanto, el problema de *mapping* restringido a grafos de tareas acíclicos es NP-completo.

El teorema anterior considera la restricción sobre grafos de tareas acíclicos y grafos de procesadores generales. Un problema aún más restrictivo es la combinación de grafos de tareas acíclicos y arquitecturas en grilla (con 4, 6 u 8 vecinos). La siguiente proposición utiliza una técnica similar al teorema de la sección anterior, y muestra que este nuevo problema sigue siendo NP-completo.

Proposición. *El problema de mapping restringido a grafos de tareas acíclicos y arquitecturas en grillas con 4 vecinos por nodo es NP-completo.*

Demostración. Construimos una transformación pseudopolinomial de 3-PARTITION al problema restringido. Sea I una instancia de 3-PARTITION, con un conjunto A de $3m$ elementos y un entero B. Sea $k = \lceil B/max_{a \in A} s(a) \rceil$. La instancia $f(I)$ consistirá del grafo G de tareas y el grafo H de procesadores, construidos como sigue: El grafo G tiene una componente $C(a)$ por cada elemento $a \in A$, como muestra la figura 2.7, y el grafo H es una grilla rectangular de $3m$ filas y kB columnas.

Del mismo modo que en el teorema de la sección anterior, esta transformación es pseudopolinomial y no es difícil verificar que hace corresponder instancias afirmativas de 3-PARTITION a instancias afirmativas del problema restringido.

Conclusión. En este capítulo se presentó el problema de *mapping* simple, y se analizó su complejidad. Es un hecho conocido que este problema pertenece a la clase de problemas *NP-*

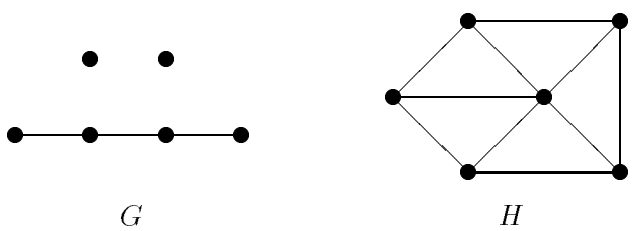


Figura 2.6: Ejemplo con $K = 3$.

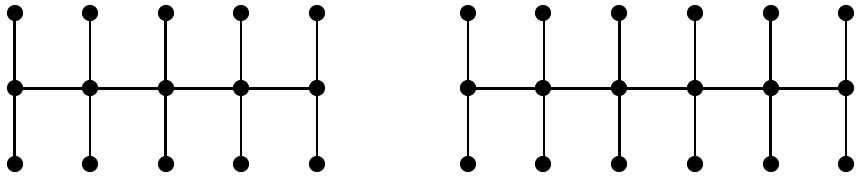


Figura 2.7: Componentes $C(a)$ utilizadas en la construcción.

Hard, pero además se mostró que sigue siendo intratable cuando se consideran las instancias que aparecen con mayor frecuencia en la práctica. En este sentido, los subproblemas sobre arquitecturas puntuales son las restricciones más naturales al problema, y para las arquitecturas en grilla (con 4, 6 y 8 vecinos) el problema continúa siendo *NP-Hard*.

Una de las estrategias inmediatas a seguir cuando se encuentra un problema de este tipo es la construcción de heurísticas, sean basadas en técnicas metaheurísticas o no. En el próximo capítulo se describe la definición e implementación de dos algoritmos heurísticos para el problema. Estos algoritmos serán de utilidad para la implementación práctica del algoritmo exacto *branch&cut*, que se describe en los capítulos subsiguientes.

Capítulo 3

Dos heurísticas

"...este claustro secular donde se incubaron mis más bellas ideas."

–Albert Einstein

La dificultad inherente de la mayoría de los problemas de optimización combinatoria torna prácticamente obligatoria la construcción de heurísticas que proporcionen soluciones razonables y requieran tiempos de ejecución bajos. Al ser improbable la existencia de algoritmos polinomiales para su resolución, los métodos heurísticos tienen una importancia decisiva en problemas "de la vida real", que se presentan en un contexto en el cual con frecuencia es más importante el tiempo de resolución que la exactitud de los resultados. Por estos motivos, resulta interesante y aplicable el estudio de algoritmos heurísticos para el problema.

Existe una segunda razón que estimula el desarrollo de estos métodos. Los algoritmos *branch&cut* se basan en el cálculo simultáneo de cotas inferior y superior del óptimo, que se acercan progresivamente a este valor. Un método que proporcione una buena solución en poco tiempo genera automáticamente (en un problema de maximización) una cota inferior acorde a la calidad de la heurística. De este modo, se tiene un muy buen punto de partida para el algorimo, lo cual puede disminuir notablemente el tiempo de ejecución, al podar grandes partes del árbol de enumeración.

Puede parecer que las heurísticas son apropiadas para problemas "reales", y que un algoritmo exacto sólo tiene interés teórico. Sin embargo, la característica fundamental de los algoritmos *branch&cut* (el cómputo simultáneo de cotas inferior y superior) los torna muy importantes para problemas prácticos. En efecto, la generación de planos de corte junto con el esquema de enumeración de un algoritmo de este tipo constituye un método que eventualmente termina, y que garantiza una solución en tiempo finito. Un algoritmo que provee solamente cotas superiores que se acercan progresivamente al óptimo no es adecuado para la resolución práctica de problemas, pero al combinar este algoritmo con un método heurístico se tiene una herramienta que puede ser muy potente: Se genera una buena solución inicial y poco tiempo después se tiene una garantía (cota superior) de su distancia al óptimo. Eventualmente estas dos cotas pueden mejorar (hallando mejores soluciones factibles o disminuyendo

la cota superior), hasta llegar al óptimo. Aunque las exigencias del tiempo de resolución obliguen a detener este proceso en poco tiempo, se tendrá, además de una solución factible, una garantía de su calidad (cfr. [Jun/92]).

En este capítulo se describe la implementación de dos métodos heurísticos para el problema de *mapping* simple. En primer lugar, se describe una implementación directa del algoritmo de rebotes simulados (SRA), y se presentan pruebas computacionales con sus parámetros. Luego, se describe una heurística híbrida basada en la técnica de *simulated annealing*, y se resumen los resultados obtenidos con este segundo método, para finalizar el capítulo con una comparación empírica entre las dos implementaciones.

3.1 Algoritmo de Rebotes Simulados

El algoritmo de rebotes simulados (SRA) es una técnica metaheurística poco conocida, desarrollada por A. Hoese [Hoe/95], que se describe en esta sección [1]. Al igual que muchos otros métodos heurísticos, este método está inspirado en la simulación de aquellos procesos físicos en los cuales se minimiza una cantidad o se logra una solución cercana a un mínimo. A lo largo de esta sección, suponemos que tenemos un problema de minimización.

Este método se basa en la analogía con el rebote de una pelota sobre una superficie escalonada, que busca puntos progresivamente más bajos, en su camino hacia una solución con poca energía potencial. Debido a la acción de la gravedad y a la pérdida gradual de energía cinética y potencial de la pelota, el proceso de rebotes se detiene en un punto que se espera sea cercano al mínimo global. Para este proceso, se considera al valor de función objetivo como la "altura" de cada escalón, y se permite que la pelota salte a una solución vecina si la energía total de la pelota lo hace posible (es decir, si es mayor que la energía potencial necesaria para alcanzar la altura de la nueva solución).

Para que el proceso de rebotes sea practicable, se debe definir una *estructura de vecindad* para el problema particular, que proporcione un conjunto de "vecinos" de una cierta solución dada. En cada paso, la pelota puede saltar desde la solución actual a una solución vecina (es decir, dentro de su estructura de vecindad). El riesgo de que la pelota "quede atrapada" en un mínimo local se reduce al considerar los saltos hacia puntos de mayor función objetivo si su energía se lo permite. Si las condiciones iniciales en las que lanza la pelota lo posibilitan, la pelota saltará los mínimos locales en busca del óptimo global.

Continuando la analogía, se deben contemplar disminuciones de la energía total debidas a las colisiones, puesto que de otro modo (si las colisiones fueran siempre elásticas) la pelota nunca se detendría. Para evitar esto, se debe considerar la pérdida de energía cuando el choque es inelástico, así como también una pequeña pérdida de energía por choques elásticos cuando se aproxima la conclusión del proceso.

Las condiciones iniciales indican el modo en el que se lanza la pelota. De acuerdo con la

[1] No se ha publicado ninguna descripción detallada del método, y la reseña de esta sección es producto de comunicaciones con su autor.

descripción anterior, estas condiciones son las siguientes:

- La solución inicial sobre la cual se deja caer la pelota.
- La energía potencial inicial E_0.
- El factor de pérdida de energía debido a colisiones inelásticas ($1 - \alpha$, con $0 < \alpha < 1$). El valor de α varía entre los extremos 0 (pelota que no rebota) y 1 (pelota que no se detiene).
- El factor de pérdida de energía debido a colisiones elásticas ($1 - \delta$, con $\alpha < \delta < 1$).

Estas condiciones conforman el conjunto de parámetros del proceso de simulación, y tienen incidencia directa en la performance del algoritmo. En la práctica, se puede construir la solución inicial automáticamente, y también pueden adoptarse los otros dos como función del parámetro α, como se verá más adelante. Esto hace que se deba considerar sólo un parámetro, simplificando el proceso de ajuste de parámetros.

3.1.1 Descripción del Algoritmo

Dada la solución inicial y la energía potencial inicial de la pelota, se comienza el proceso de rebotes. En cada paso, se calculan los vecinos de la solución actual, considerando la posibilidad de saltar hacia uno de ellos. Si la energía total de la pelota (energía cinética y potencial) no es suficiente para alcanzar a ninguno de los vecinos, se interrumpe el proceso, dado que la pelota ha quedado atrapada en la solución actual.

Si dicha energía es mayor que la energía potencial de alguno de los vecinos, la pelota salta al vecino, que pasa a considerarse como punto de rebote de la siguiente iteración. Si la energía del rebote es muy grande (mayor que el coeficiente de elasticidad) entonces la colisión produce deformación y parte de la energía se pierde (choque inelástico). Por el contrario, si la energía del choque no es tan grande como para producir deformaciones, sólo una pequeña parte de la energía se pierde (producida por fricciones y otros factores). Esta pequeña pérdida de energía se introduce en el algoritmo como una disminución lineal en un factor de $1 - \delta$ (con $\delta \in [0, 1]$, cercano a 1). Debe notarse que este factor, aunque pequeño, es necesario puesto que de lo contrario la pelota no dejaría de rebotar nunca.

Llamemos E a la energía de la pelota en el actual punto de rebote, y $F(x)$ al valor de la función objetivo en el punto x. Con esto, se producen tres casos posibles con relación a un vecino x' particular:

Caso 1: La solución vecina x' no puede ser el nuevo punto de rebote, dado que su altura $F(x')$ es mayor que la altura máxima δE que puede alcanzar la pelota. En este caso, se pasa a considerar al siguiente vecino, y si no hay otro vecino, el método se detiene.

Caso 2: La nueva solución se encuentra por debajo de la altura máxima δE, pero sin provocar choque inelástico (es decir, por sobre la altura límite αE para choques inelásticos).

El resultado es un choque inelástico y no se pierde energía adicional al pasar al nuevo punto de rebote.

Caso 3: La nueva solución se encuentra por debajo de la altura límite para choques elásticos ($F(x') < \alpha E$). La diferencia de alturas hace que se produzca un choque inelástico, con lo cual la pelota pierde energía, pasando a la nueva solución con una energía de αE.

De acuerdo con lo anterior, puede verse que se tienen tres franjas. Si la nueva solución está por arriba de δE, entonces es inalcanzable por la pelota. Si se encuentra entre αE y δE, se produce un choque elástico, y si se encuentra por debajo de αE el choque es inelástico, perdiendo energía. Se realiza este proceso con todos los vecinos de la solución actual, que se recorren en forma arbitraria. El siguiente pseudocódigo resume la descripción anterior:

```
s <- solucionInicial();
Repetir

    Mientras haya un vecino s' de s aun no visitado:
        Si F(s') < alfa * E:
            E <- alfa*E;
            s <- s';
            salir;
        Si F(s') < delta * E:
            E <- delta*E;
            s <- s';
            salir;
    Fin (para)
    Si no se modifico la solucion, terminar.

Fin
```

Reducción de Parámetros. Se mencionaron cuatro parámetros como condiciones iniciales del algoritmo. El algoritmo general tiene estos cuatro parámetros, para los cuales deben buscarse valores apropiados. Al costo de perder generalidad en su selección, se pueden escribir estos parámetros en función de uno de ellos, la reducción por choques inelásticos α. Este parágrafo muestra una forma posible de hacerlo, que produce buenos resultados en la práctica.

En primer lugar, puede tomarse la energía inicial E_0 como función del parámetro α y la altura de la solución inicial:

$$E_0 = \alpha^{-1} F(x_0)$$

siempre que $0.7 < \alpha < 0.99$. Por otra parte, el parámetro δ se puede fijar como

$$\delta = (9 + \alpha)/10$$

Por último, la solución inicial puede calcularse en forma automática, mediante una heurística de construcción aleatoria, o utilizando procedimientos más sofisticados. Con esto, los parámetros

del algoritmo se reducen a uno solo, que indica la forma en la cual se producen los rebotes. Valores altos de α determinan una pelota que rebota en mayor medida.

Aceleración de la Convergencia. Es posible lograr que la convergencia se acelere introduciendo una variación al esquema anterior. Cuando se produce un rebote frustrado (caso 1 de la descripción del método), se puede reducir la energía de la pelota. Esta reducción se realiza cuando la energía de la pelota no permite alcanzar una solución vecina, y debe permitir la exploración completa del espacio de soluciones. Cuando se produce un choque frustrado, la energía de la pelota disminuye en β, donde

$$\beta < \frac{E - Ep}{\Phi}$$

siendo Ep la energía potencial de la solución actual y Φ la cantidad de vecinos. Esta reducción detiene el proceso de rebotes en una cierta iteración si se exploró por completo el espacio de soluciones vecino sin hallar una solución que pudiera ser alcanzada.

Implementación Práctica. Para la implementación práctica de la heurística, se debe contar con una estructura de vecindad para el problema particular y una forma de hallar una solución inicial. La calidad de la solución depende de la solución inicial y del parámetro α. En una implementación de esta heurística, se pueden considerar las siguientes variantes:

- Se puede ejecutar varias veces, con soluciones iniciales distintas, para realizar una exploración mayor del espacio de soluciones. Esto es posible dado que el algoritmo es muy eficiente, y no es en general muy costosa la ejecución desde distintas soluciones iniciales.
- Se puede seguir la trayectoria de la pelota, guardando todas las soluciones por las que pasa. De esta forma, se puede obtener un conjunto de soluciones cercanas al óptimo, e incluso el óptimo que puede haber sido saltado por la pelota en virtud de su energía en el momento de alcanzarlo.

Adaptación al problema de *mapping* simple. Se implementó la heurística siguiendo estos lineamientos. En particular, se ejecuta la simulación con varias soluciones iniciales, lo cual permite abarcar un mayor espacio de soluciones sin incurrir en costos importantes. En cada iteración se genera una solución aleatoria que se utiliza como punto inicial para el proceso. Esta solución se construye asignando un procesador a cada tarea de modo aleatorio.

El ajuste más importante que se debió realizar fue el cálculo de la estructura de vecindad. Dada una solución s, un primer vecindario posible es considerar todas las soluciones que se obtienen intercambiando los procesadores de un par de tareas. Todos los posibles intercambios de a pares conforman los vecinos de la solución. Sin embargo, este vecinadario tiene $O(n^2)$ elementos, con lo cual puede llegar a ser muy costoso recorrer todas las soluciones vecinas en cada iteración, y se utiliza una versión reducida de esta estructura.

Con el objetivo de mantener una cantidad reducida de vecinos, se adoptó el siguiente esquema: En cada punto de rebote, antes de considerar a los vecinos, se elige una tarea i aleatoriamente y se considera su procesador. Cada vecino se obtiene tomando otra tarea e intercambiando el procesador al cual está asignada con la tarea i inicial. Llamemos $N_i(s)$ al

conjunto de soluciones vecinas generadas por este procedimiento. De esta forma, se tienen $|N_i(s)| = n-1$ soluciones vecinas. Además, este modo de considerar los vecinos permite pasar de una solución a cualquier otra solución, con lo cual no excluye la exploración de ningún elemento del espacio de soluciones.

Proposición. *Sean s y s' dos soluciones. Entonces, existe una cadena de soluciones $s = s_0, s_1, ..., s_n = s'$ tal que $s_{t+1} \in N_{i_t}(s_t)$ para alguna tarea i_t $(0 \leq t < n)$.*

Demostración. Sea $d(s, s')$ la cantidad de tareas de s que no están ubicadas en el mismo procesador que s'. Procedemos por inducción en $d(s, s')$. Si $d(s, s') = 0$, entonces $s = s'$, con lo cual la cadena buscada tiene un sólo elemento $(n = 0)$. Para el paso inductivo, supongamos que $d(s, s') > 0$. Entonces, existe una tarea i asignada al procesador k en s y al procesador $k' \neq k$ en s'. Sea j la tarea que está asignada al procesador k' en s, y llamemos s_1 a la solución obtenida a partir de s intercambiando las tareas i y j. Se tiene que $s_1 \in N_i(s)$, y además $d(s_1, s') < d(s, s')$, dado que ahora la tarea i está en el mismo procesador en s_1 y en s' (y las tareas distintas de i y de j no cambian de procesador). Por hipótesis inductiva, existe una cadena de soluciones vecinas $s_1, s_2, ..., s_n = s'$, con lo cual se forma la cadena $s = s_0, s_1, ..., s_n = s'$.

3.2 SRA: Resultados computacionales

El éxito de la aplicación de una heurística para un problema particular depende, en muchos casos, de los valores de los parámetros que se utilicen. En esta sección, se describen las pruebas realizadas con el objetivo de evaluar el comportamiento del algoritmo de rebotes simulados (SRA) bajo distintas combinaciones de valores de sus parámetros. Con estas pruebas solamente se busca conocer la calidad de las soluciones halladas en función de los parámetros, sin realizar comparaciones con otras heurísticas. Esas comparaciones se describen en una sección posterior.

De acuerdo con la descripción anterior, esta heurística requiere cuatro parámetros, pero estos parámetros se pueden reducir al factor de rebotes α, si la solución inicial se genera aleatoriamente ($\alpha \in [0, 1]$, donde 0 indica una pelota que no rebota, y 1 una pelota que no se detiene). Por otra parte, se realizan varias iteraciones del proceso de rebotes, lanzando la pelota cada vez desde una solución distinta. La cantidad de iteraciones realizadas se toma como el segundo parámetro de la heurística. Este segundo parámetro es importante, dado que una cantidad demasiado baja de iteraciones hace que recorra una porción pequeña del espacio de soluciones, mientras que un valor muy alto aumenta el tiempo de ejecución. Podría pensarse que ocurre una situación similar dentro de cada iteración con el parámetro α, dado que una pelota con factor de rebotes alto tarda más tiempo en detenerse.

Calidad de la solución. El primer conjunto de pruebas busca estimar la calidad de las soluciones halladas en función de los parámetros. Se realizaron estas pruebas sobre la instancia STR4, de 15 nodos (cfr. sección 7.1). Una implementación preliminar del algoritmo *branch&cut* no pudo resolver esta instancia, obteniendo una diferencia muy grande entre las cotas inferior y superior. Por estos motivos, se hicieron muchas pruebas sobre esta instancia,

para buscar parámetros de la heurística que arrojaran mejores resultados. La tabla 3.1 resume los resultados, mostrando el promedio del valor de las soluciones halladas luego de 867 ejecuciones para cada combinación de los parámetros.

	Parámetro α				
Iteraciones	**0.95**	**0.96**	**0.97**	**0.98**	**0.99**
10	22.230	22.574	22.720	23.055	23.486
20	22.822	23.156	23.242	23.566	23.877
30	23.124	23.440	23.521	23.806	24.091
40	23.348	23.618	23.717	23.957	24.199
50	23.484	23.748	23.823	24.050	24.293
60	23.597	23.839	23.910	24.133	24.364
70	23.701	23.903	24.016	24.191	24.429
80	23.749	23.965	24.065	24.241	24.478
90	23.817	24.013	24.108	24.276	24.523
100	23.869	24.042	24.153	24.308	24.566

Tabla 3.1: Promedio de las soluciones obtenidas en función de la cantidad de iteraciones y del parámetro α

Como puede verse en esta tabla, para un valor de α fijo, la calidad de la solución aumenta a medida que aumenta la cantidad de iteraciones, aunque este aumento se hace progresivamente más pequeño. De la misma manera, para una cantidad fija de iteraciones, la calidad de la solución aumenta cuanto mayor es el valor de α. Estas pruebas confirman la idea intuitiva que se tiene, dado que valores mayores de α y de la cantidad de iteraciones hacen que se recorra una porción mayor del espacio de soluciones, obteniendo mejores soluciones. Se observa el mismo comportamiento sobre otras instancias.

Tiempo de ejecución. Estas pruebas, sin embargo, muestran solamente la calidad de la solución, sin considerar el tiempo de ejecución necesario para alcanzar estos valores. El segundo conjunto de pruebas mide el tiempo de ejecución para distintos valores de los parámetros, resumidas en la tabla 3.2. Esta tabla muestra el promedio del tiempo en segundos, para la instancia STR4, sobre 30 ejecuciones para cada combinación de los parámetros.

Esta tabla muestra cómo el aumento de la cantidad de iteraciones provoca un aumento prácticamente proporcional en el tiempo de ejecución. Por otra parte, el aumento intuitivo del tiempo que se esperaría para valores grandes de α no se observa en la tabla. Por ejemplo, para 60 iteraciones, el mayor tiempo se obtuvo con $\alpha = 0.96$, por encima del tiempo logrado con $\alpha = 0.99$, a pesar que en este caso la pelota tiene un mayor factor de rebote, y se espera que se encuentre más tiempo recorriendo soluciones.

Se realizaron pruebas más intensivas, para determinar el tiempo de ejecución bajo distintos valores de α, esta vez con una mayor cantidad de ejecuciones y en un rango más amplio. Este tercer conjunto de pruebas se realizó fijando en 50 la cantidad de iteraciones, y se utilizaron seis instancias distintas. La tabla 3.3 muestra los resultados obtenidos, indicando el promedio del tiempo sobre 150 ejecuciones de cada instancia para cada valor del parámetro α.

Como muestra la tabla 3.3, el tiempo tiende a aumentar a medida que aumenta el factor

	Parámetro α				
Iteraciones	**0.95**	**0.96**	**0.97**	**0.98**	**0.99**
10	0.1503	0.1626	0.1526	0.1473	0.1510
20	0.2926	0.3353	0.3000	0.2820	0.3040
30	0.4520	0.4933	0.4650	0.4360	0.4486
40	0.5933	0.6556	0.6220	0.5776	0.6150
50	0.7476	0.8146	0.7606	0.7146	0.7493
60	0.8603	0.9920	0.9010	0.8803	0.9183
70	1.0423	1.1666	1.0626	1.0206	1.0503
80	1.2336	1.3203	1.2333	1.1520	1.2096
90	1.3543	1.4756	1.3723	1.2890	1.3496
100	1.5120	1.6590	1.5166	1.4690	1.4870

Tabla 3.2: Promedio del tiempo de ejecución (en sg.) en función de la cantidad de iteraciones y del parámetro α.

α. Sin embargo, este aumento no es progresivo y presenta irregularidades. Por ejemplo, en la instancia PRB9 el tiempo aumenta progresivamente hasta $\alpha = 0.94$, momento en el cual disminuye, llegando a los mismos valores con $\alpha = 0.99$ que con $0.87 \leq \alpha \leq 0.89$. Sin embargo, en las instancias STR7 y DC3, el tiempo aumenta dramáticamente cuando $\alpha \geq 0.95$.

Puede concluirse, entonces, que la calidad de la solución aumenta a medida que el parámetro α se aproxima a 1, pero esto trae aparejado un aumento del tiempo de ejecución. Este aumento en el tiempo es irregular, aunque en algunos casos puede ser muy importante.

Tiempo vs. tamaño del problema. Por último, se realizaron pruebas para determinar la variación del tiempo de ejecución con relación a la cantidad de nodos del problema, dado que se observan grandes diferencias entre distintos problemas. Como el vecindario de una solución tiene un tamaño que depende de la cantidad de nodos del problema, se espera un aumento del tiempo de ejecución con relación a la cantidad de nodos. Se realizaron pruebas sobre instancias con estructura similar, y los resultados se muestran en la tabla 3.4, que indica el promedio del tiempo de ejecución para 150 corridas, con 50 iteraciones cada una. Esta tabla muestra el aumento esperado en el tiempo de ejecución a medida que crece el tamaño del problema.

En concusión, las pruebas realizadas muestran un comportamiento que responde a las estimaciones que se realizaron a priori. Con esto, se finaliza la descripción de la heurística SRA. La siguiente sección describe la segunda heurística implementada, las pruebas realizadas y una comparación empírica entre los dos métodos.

3.3 Chained Local Optimization (CLO)

Esta heurística es una adaptación de la técnica metaheurística *Chained Local Optimization* (CLO), introducida en [Mar/96]. Combina las ideas de *simulated annealing* con procedimientos de búsqueda local, y ha arrojado muy buenos resultados para el problema del viajante

α	STR4	STR6	STR7	DC2	DC3	PRB9
0.80	0.254	0.483	0.972	0.177	1.826	0.113
0.81	0.269	0.480	1.026	0.185	1.883	0.127
0.82	0.258	0.516	1.040	0.180	1.812	0.148
0.83	0.264	0.506	0.872	0.302	2.012	0.174
0.84	0.320	0.527	0.934	0.284	1.784	0.148
0.85	0.278	0.503	0.956	0.207	1.929	0.197
0.86	0.327	0.499	1.228	0.204	1.830	0.186
0.87	0.335	0.536	0.876	0.211	1.738	0.198
0.88	0.375	0.636	0.900	0.216	1.783	0.206
0.89	0.383	0.560	0.880	0.238	1.787	0.204
0.90	0.407	0.177	0.997	0.244	1.799	0.219
0.91	0.523	0.751	1.624	0.289	1.696	0.218
0.92	0.551	0.887	0.977	0.323	1.699	0.265
0.93	0.628	1.017	1.088	0.369	1.904	0.272
0.94	0.683	1.447	1.358	0.441	1.791	0.288
0.95	0.754	1.549	1.712	0.498	1.798	0.257
0.96	0.813	1.841	2.448	0.620	1.715	0.237
0.97	0.768	2.049	3.406	0.656	1.927	0.218
0.98	0.731	1.860	4.180	0.551	2.185	0.201
0.99	0.747	1.532	3.698	0.471	5.585	0.206

Tabla 3.3: Promedio del tiempo de ejecución (en sg.)
para seis instancias de prueba

de comercio y para instancias del problema de partición de grafos. Se describe primero la técnica, para puntualizar los aspectos particulares de la implementación realizada.

Esta técnica requiere la definición de un procedimiento de búsqueda local. Dado un vecindario $N(s) \subseteq S$ para cada solución $s \in S$, un procedimiento típico de búsqueda local mantiene una solución actual s. Esta solución que se reemplaza por otra $s' \in N(s)$ si $f(s') > f(s)$, y el procedimiento se detiene cuando $f(s) \geq s'$, $\forall s' \in N(s)$ (en este caso, se dice que ha llegado a un *óptimo local* con respecto al vecindario definido). Llamemos `busquedaLocal(s)` a este procedimiento, que parte de la solución s y llega a un óptimo local.

La técnica de *simulated annealing* no saca partido de la búsqueda local, sino que considera a todas las soluciones en su búsqueda. En lugar de esto, podemos hacer que se recorran sólo los óptimos locales, y sobre esta idea se basa la técnica CLO. Supongamos que la solución actual es localmente óptima, y apliquemos una perturbación que la modifique en forma significativa. En *simulated annealing* standard, se compara esta nueva solución con la inicial, para decidir si la reemplaza o no. Sin embargo, puede ser beneficioso mejorar esta nueva solución mediante el procedimiento de búsqueda local. De esta forma, se obtiene un nuevo óptimo local, que ahora se compara con la solución inicial.

De esta forma, *el espacio de soluciones de CLO está formado por los óptimos locales del espacio inicial.* Esto produce el efecto de "suavizar el paisaje", eliminando los cambios

α	DF1 ($n = 12$)	DF4 ($n = 15$)	DF6 ($n = 16$)	DF3 ($n = 18$)	DF5 ($n = 20$)	DF2 ($n = 30$)
0.80	0.162	0.232	0.247	0.343	0.417	1.204
0.83	0.165	0.263	0.306	0.339	0.522	1.357
0.85	0.192	0.244	0.316	0.300	0.504	1.778
0.87	0.230	0.291	0.334	0.320	0.420	1.147
0.90	0.293	0.372	0.454	0.356	0.458	1.209
0.93	0.366	0.490	0.694	0.632	0.622	1.298
0.95	0.351	0.690	0.834	0.854	0.956	1.370
0.97	0.328	0.656	0.854	1.222	1.503	2.151
Total:	**2.087**	**3.305**	**4.039**	**4.336**	**5.402**	**11.514**

Tabla 3.4: Promedio del tiempo de ejecución (en sg.)
para seis instancias distintas, entre 12 y 30 nodos

abruptos[2]. La figura 3.1 muestra el pseudocódigo de esta técnica. En esta figura, `f(s)` denota el valor de la función objetivo de la solución `s`, y `To` y `t` son parámetros del algoritmo.

El procedimiento `granModificacion()` debe constituir una modificación más importante que la utilizada en el vecindario de la búsqueda local, para que la solución `s'` no quede atrapada en las cercanías del mínimo local `s` (cfr. [Mar/96]). Como puede verse en la figura, el método sigue los lineamientos de *simulated annealing*: Cada ejecución del ciclo exterior de denomina una *época*, y se considera una *temperatura* `T`, que se reduce luego de cada época. Además de los conceptos básicos de la técnica *simulated annealing*, se introduce el procedimiento de búsqueda local luego de la generación de la solución inicial y de los vecinos de la solución. Se describen a continuación los ajustes que fueron necesarios para la implementación de esta técnica.

Búsqueda local. El procedimiento de búsqueda local está determinado por la elección del vecindario. Dada una solución y una tarea i, sus vecinos se obtienen intercambiando la tarea i con todas las otras tareas j (la tarea i pasa a ocupar el procesador en el que estaba j y la tarea j ocupa el procesador que i deja libre). La tarea i se elige aleatoriamente en cada iteración.

Modificación. El procedimiento `granModificacion(s)` realiza una perturbación más importante de la solución `s` que recibe como parámetro. Se selecciona aleatoriamente un conjunto T de 15 tareas, y se calcula el conjunto W de procesadores sobre los cuales las tareas de T están asignadas en `s`. La modificación se lleva a cabo reasignando aleatoriamente las tareas de T sobre los procesadores de W. Es decir, se toma un conjunto de tareas, intercambiando sus procesadores simultáneamente. Con esto se logra una alteración más importante que el intercambio utilizado en la búsqueda local. La cantidad de 15 tareas resultó adecuada para las instancias de prueba consideradas, pero debería aumentarse para tratar instancias de mayor tamaño.

[2] Los autores que presentan esta técnica metaheurística (cfr. [Mar/96]) aplicaron esta técnica al problema del viajante de comercio, y observaron que en algunos casos este nuevo espacio de soluciones tiene un sólo óptimo local.

```
s <- solucionInicial();
s <- busquedaLocal(s);
T <- To;

Mientras( no se cumpla la condicion de terminacion )

    Repetir t veces:

        s'  <- granModificacion(s);
        s'' <- busquedaLocal(s');
        df  <- f(s'') - f(s);

        si df <  0 : s <- s'';
        si df >= 0 : s <- s'' con probabilidad exp(-df/T);

    Fin( repetir );
    Reducir T;

Fin( mientras );
```

Figura 3.1: CLO, esquema general.

Condición de terminación. Se detiene el procedimiento cuando se llega a una temperatura mínima, o bien cuando luego de r épocas no se mejora la mejor solución hallada hasta el momento. Otras posibilidades podrían ser un límite superior a la cantidad de iteraciones o un tiempo máximo de ejecución, pero no se tuvieron en cuenta estas opciones, dado que se obtuvieron buenos resultados con el primer criterio.

Debe notarse que este procedimiento es muy similar a la heurística propuesta por Bokhari (cfr. [Bok/81]), que consta de una fase de mejora con un vecindario similar, seguida de un "salto probabilistico" esencialmente igual al procedimiento `granModificacion(s)`. En el artículo citado, se reportan muy buenos resultados con este procedimiento, y las pruebas realizadas confirman estas tendencias.

3.4 CLO: Resultados computacionales

En la sección 3.2 se describieron los resultados de las pruebas realizadas con el método de rebotes simulados. Se resumen ahora los resultados de las pruebas realizadas con la heurística CLO, para finalizar el capítulo con una comparación empírica entre ambas heurísticas.

El objetivo de estas pruebas fue determinar el comportamiento del método bajo distintas combinaciones de valores para sus parámetros. Este método es probabilístico, a diferencia del método SRA, para el cual su naturaleza determinística permite realizar un análisis más

preciso de su comportamiento. Las pruebas realizadas contribuyeron a encontrar parámetros adecuados para su empleo en el algoritmo *branch&cut*.

Iteraciones por época. La heurística implementada ejecuta una cierta cantidad t de iteraciones por cada época. En el primer conjunto de pruebas, se realizaron ejecuciones modificando el valor de t, para observar los valores alcanzados y el tiempo de ejecución. Se realizaron las pruebas sobre 20 problemas test, y la tabla 3.5a muestra la suma de los valores obtenidos, y el tiempo total para las 20 instancias. Para estas pruebas, la temperatura inicial se calcula en forma automática, y se reduce en cada época en un factor de 0.95, hasta llegar a un mínimo de 0.01, o hasta que se encuentren 5 epocas sin mejorar la mejor solución.

t	Solución	Tiempo (sg.)
10	527	180.54
20	529	583.07
30	529	1118.51
40	531	1205.71
50	533	1514.23
60	531	1505.93
70	534	2216.42
80	531	2455.76
90	532	2581.42
100	534	3280.65

r	Solución	Tiempo (sg.)
2	529	571.92
3	529	983.73
4	529	1386.09
5	533	1372.69
6	531	2298.62
7	531	1712.96
8	532	2250.16
9	534	2594.50
10	535	2739.01
11	535	3430.05

Tabla 3.5: Resultados variando (a) el parámetro t y (b) el parámetro r

Puede verse en esta tabla que la calidad de las soluciones tiende a aumentar levemente a medida que se aumenta la cantidad de iteraciones por época. Este aumento no es muy grande, y se realiza con altibajos. Sin embargo, el tiempo de ejecución crece progresivamente, hasta llegar a valores muy altos en comparación con los iniciales. En suma, el incremento de este parámetro ocasiona una pequeña mejora de la solución, pero el tiempo necesario para hacerlo puede crecer excesivamente en comparación con el aumento logrado en la calidad de la solución.

Épocas sin mejora. La heurística realiza t iteraciones por época, luego de las cuales se disminuye la temperatura y se pasa a la siguiente época. El criterio de terminación implementado detiene el método si se ha llegado a una cierta temperatura, o cuando han pasado r épocas sin mejorar la mejor solución. Este segundo grupo de pruebas modifica progresivamente el parámetro r, obteniendo los valores que se muestran en la tabla 3.5b. Los parámetros utilizados son los mismos que en el caso anterior (salvo el parámetro r, que se modifica), realizando 30 iteraciones por época. Se observa un comportamiento similar al caso anterior. Nuevamente presenta altibajos, aunque puede observarse una cierta tendencia en esta tabla: La calidad de la solución aumenta, pero este aumento es muy leve en comparación con el tiempo de ejecución cada vez mayor.

Reducción de la temperatura. Luego de ejecutar una época, la temperatura disminuye en un factor β. En las implementaciones de *simulated annealing* este parámetro se fija entre 0.8 y 0.99, y se realizaron pruebas para determinar qué valor dentro de este rango es el más conveniente. La tabla 3.6 muestra los resultados, en los cuales no se observa un compor-

tamiento definido. Tanto la calidad de la solución como el tiempo de ejecución no parecen depender directamente de este parámetro, lo cual podría explicarse por la gran cantidad de factores que se tienen en cuenta en la heurística.

β	Solución	Tiempo (sg.)
0.80	527	537.52
0.81	529	556.04
0.82	526	576.53
0.83	531	868.68
0.84	530	641.09
0.85	528	750.87
0.86	528	778.68
0.87	529	739.34
0.88	525	847.36
0.89	527	550.27
0.90	527	418.51
0.91	528	655.76
0.92	528	736.04
0.93	528	627.36
0.94	529	671.53
0.95	529	632.47
0.96	527	585.16
0.97	528	595.21
0.98	530	786.15
0.99	529	644.45

Tabla 3.6: Resultados variando el parámetro β

En resumen, las pruebas realizadas muestran que la calidad de la solución varía levemente en función de los parámetros, haciendo que la elección de los parámetros iniciales pueda afectar en poco la salida del método (siempre que estos parámetros tengan valores "razonables"). Por otra parte, se observó que el tiempo de ejecución puede aumentar mucho si se eligen combinaciones "agresivas" para los parámetros, con lo cual deben elegirse cuidadosamente sus valores, para evitar tiempos excesivos.

3.5 Comparación SRA-CLO

Como se mencionó, el objetivo de una heurística es buscar soluciones buenas en tiempo razonable, lo cual es de importancia fundamental en la práctica y puede contribuir en forma decisiva como cota primal en un algoritmo *branch&cut*. En este sentido, debe decidirse cuál de las dos heurísticas se agregará al algoritmo, si es que alguna de ellas prevalece siempre sobre la otra.

El último grupo de pruebas realizado sobre las heurísticas consistió en una comparación entre ambas, a fin de decidir si alguna de ellas es mejor. Se ejecutaron ambos métodos so-

bre el grupo de problemas test. Para el método CLO, se tomaron 30 iteraciones por época, con un factor de reducción de temperatura de 0.95. La temperatura inicial se calculó automáticamente, y se especificó como criterio de terminación el paso de 5 épocas consecutivas sin mejorar la mejor solución. Por su parte, para la heurística SRA, se ejecutaron $5n^2$ iteraciones (siendo n la cantidad de nodos del problema), con $\alpha = 0.99$.

La tabla 3.7 resume los resultados. Las columnas $e(G)$ y $e(H)$ indican la cantidad de ejes del grafo de tareas y el grafo de procesadores, respectivamente. Para cada método se reporta la solución hallada y el tiempo total de cómputo para cada instancia.

Problema	Nodos	$e(G)$	$e(H)$	CLO		SRA	
STR1	9	36	14	14	0.219 sg.	14	0.164 sg.
STR2	6	15	8	8	<0.001 sg.	8	0.054 sg.
STR3	15	60	26	26	3.626 sg.	26	5.714 sg.
STR4	15	60	26	25	3.296 sg.	25	5.604 sg.
STR5	20	80	36	**34**	20.219 sg.	31	18.021 sg.
STR6	20	80	37	**36**	17.692 sg.	32	18.406 sg.
STR7	25	100	48	**45**	50.879 sg.	36	63.186 sg.
STR8	25	100	51	**47**	39.780 sg.	36	63.296 sg.
DF1	12	25	18	18	1.098 sg.	18	1.428 sg.
DF2	30	60	60	**38**	84.450 sg.	26	133.516 sg.
DF3	18	30	36	**21**	5.549 sg.	19	9.395 sg.
DF4	15	30	30	20	2.692 sg.	20	3.846 sg.
DF5	20	35	40	**26**	9.560 sg.	22	14.395 sg.
DF6	16	35	32	**27**	4.285 sg.	25	5.109 sg.
DF7	20	40	40	**29**	23.131 sg.	24	104.010 sg.
DF8	18	40	36	**30**	15.879 sg.	23	10.054 sg.
DC1	18	14	36	**14**	7.747 sg.	12	8.076 sg.
DC2	15	14	30	**14**	3.406 sg.	12	3.076 sg.
DC3	35	30	70	**29**	332.197 sg.	16	176.263 sg.
DC4	40	30	80	**28**	317.582 sg.	12	233.516 sg.

Tabla 3.7: Comparación entre SRA y CLO.

En todos los casos, la solución hallada por el método CLO fue superior o igual a la solución encontrada por SRA. Esto sucede a pesar de que en la mayoría de los casos la segunda heurística dispuso de un mayor tiempo de ejecución (se eligieron los parámetros para que esto sucediera). Debe notarse, además, que el tiempo de ejecución para instancias individuales no superó los 7 minutos para instancias de hasta 40 nodos (de todos modos, pueden ajustarse los parámetros para modificar este tiempo).

Conclusión. Se presentó la implementación de dos métodos heurísticos para el problema y se resumieron las pruebas computacionales realizadas con ellos para (a) ajustar sus parámetros y (b) comparar su performance. Se obtuvieron buenos resultados con una adaptación de la técnica metaheurística CLO, similar al algoritmo de "saltos probabilisticos". El desarrollo y testeo de heurísticas para el problema tiene interés dado que generan cotas inferiores del óptimo y pueden agregarse al algoritmo *branch&cut*.

Capítulo 4

Un problema dual combinatorio

"En la parte inferior del escalón, hacia la derecha, vi una pequeña esfera tornasolada, de casi intolerable fulgor. Al principio la creí giratoria; luego comprendí que ese movimiento era una ilusión producida por los vertiginosos espectáculos que encerraba. El diámetro del Aleph sería de dos o tres centímetros, pero el espacio cósmico estaba ahí sin disminución de tamaño."

–Jorge Luis Borges

Consideremos el problema $z = \max \{cx : x \in S\}$, donde $S = \{x \in Z_+^n : Ax \leq b\}$. El conjunto S contiene las soluciones factibles del problema, que están dadas por los puntos enteros de $\{x : Ax \leq b\}$. Las técnicas heurísticas habituales recorren soluciones de este conjunto S, reteniendo las mejores soluciones (aquellas con mayor función objetivo). Como se trata de un problema de maximización, estas soluciones proporcionan cotas inferiores del óptimo $z = cx^*$, y en general no es posible saber cuan lejos se hallan de este valor. Esta es una característica que se observa en la gran mayoría de las implementaciones de heurísticas, siendo la experimentación la única herramienta para estimar la bondad de las soluciones halladas. Se puede obtener una medida precisa de la performance media de la heurística si las pruebas se aplican en problemas con óptimos conocidos.

Sin embargo, con la experimentación sólo se puede formar una idea general del rendimiento, lo cual no asegura que la misma performance se repita en cada problema particular. Es decir, puede conocerse en promedio la calidad de las soluciones, pero puede suceder que la solución arrojada en instancias individuales no resulte de la misma calidad que el promedio, y no es posible saberlo si no se conoce el óptimo del problema ni se tienen cotas superiores de este valor. Precisamente es ésta la situación en problemas puntuales "de la vida real", cuyos óptimos no son conocidos o son difíciles de calcular. Cuando se aplica un procedimiento heurístico para estos problemas, no se puede conocer la bondad de la solución hallada, más allá de las pruebas y comparaciones hechas con ejemplos conocidos.

Supongamos que tenemos una instancia particular del problema, cuya solución óptima no

es conocida, y que es suficientemente grande como para tornar difícil computacionalmente su resolución exacta. Podemos utilizar alguna técnica heurística para hallar una solución factible en un lapso corto, pero no tenemos forma de saber si la solución hallada es el óptimo, o de obtener alguna garantía que proporcione una idea de su "distancia" al óptimo. Una posibilidad para conocer la "bondad" de la solución hallada es contar con mecanismos para generar cotas superiores del óptimo del problema, con lo cual se tiene una acotación de este valor, en términos de la solución heurística (cota inferior) y las cotas superiores halladas. De esta forma, se puede tener una idea de la calidad de la solución encontrada para la instancia individual, lo cual puede ser de gran utilidad.

En este capítulo se describe la búsqueda de un procedimiento para generar cotas superiores del óptimo del problema de *mapping*, que surgió a partir de un problema dual de naturaleza combinatoria. Este dual se basa en los estudios de Michael Jünger y Volker Kaibel (cfr. [Jun/95]) del problema de asignación cuadrática (QAP). Se obtiene un problema dual para el problema de *mapping*, y se demuestra que en algunos casos permite no sólo generar cotas superiores, sino que su óptimo coincide con el óptimo del problema original (es decir, constituye un dual fuerte). Sin embargo, el dual obtenido tiene gran tamaño, con lo cual sólo es aplicable a problemas pequeños.

4.1 Formulación alternativa del problema de *mapping*

Dado un problema entero IP, definido por $z = \max\{cx : x \in S\}$ $(S \subseteq Z_+^n)$, un *problema dual* de IP es un problema $z_D = \min\{dx : x \in S_D\}$, que verifica que $cx \leq dy$, para todo $x \in S$ y para todo $y \in S_D$. De esta forma, todas las soluciones factibles del dual proporcionan cotas superiores de z (con lo cual $z \leq z_D$). El problema dual se dice *fuerte* si $z = z_D$, y se dice *débil* si $z < z_D$.

Si se puede encontrar un problema dual del original, cualquier heurística aplicada a este problema que recorra soluciones de S_D proporcionará cotas superiores de z. Con esto, dado un problema particular, se ejecutan las heurísticas primal y dual, obteniendo soluciones $\bar{x} \in S$ y $\bar{x}_D \in S_D$, de modo tal que $c\bar{x} \leq z \leq d\bar{x}_D$. Con esto, se tiene una acotación del óptimo y una garantía de la distancia entre $c\bar{x}$ y z.

A continuación, se describe la obtención de un problema dual de naturaleza combinatoria. En primer lugar, se muestra cómo puede transformarse el problema de *mapping* en un problema de clique de peso máximo, en base a estudios del problema de asignación cuadrática. Luego, se muestra una segunda transformación, que lleva este problema a un problema de clique máxima (sin pesos), de modo tal que el problema de coloreo sobre el grafo resultante constituye un dual débil del problema original. Se prueba, además, que en algunos casos este problema es efectivamente un problema dual fuerte, y se cierra el capítulo con experimentos computacionales sobre este dual.

4.1.1 Transformación en un problema de clique de peso máximo

El problema de *mapping* simple es un caso particular del problema de asignación cuadrática (QAP, *quadratic assignment problem*). El problema de asignación cuadrática es un problema clásico de optimización combinatoria (cfr. [Jun/96]), con la siguiente formulación[1]:

$$\begin{aligned}
&\max \sum_{1\leq i,j,k,l\leq n} g_{ijkl}x_{ij}x_{kl} + \sum_{1\leq i,j\leq n} h_{ij}x_{ij} \\
&\sum_{j=1}^{n} x_{ij} = 1 \quad 1 \leq i \leq n \\
&\sum_{i=1}^{n} x_{ij} = 1 \quad 1 \leq j \leq n \\
&x_{ij} \in \{0,1\}
\end{aligned}$$

Esta formulación es muy similar a la del problema de asignación. Dadas n tareas, n procesadores y una matriz d que indica el costo de asignar cada tarea sobre cada procesador, el problema de asignación consiste en ubicar las tareas sobre los procesadores de modo de optimizar los costos. Las soluciones factibles de este problema ubican cada tarea sobre un procesador (de modo tal que cada procesador recibe sólo una tarea), y se denominan habitualmente *asignaciones*. Si la variable y_{ik} indica que la tarea i se asigna sobre el procesador k, el problema de asignación está definido por el siguiente programa lineal entero:

$$\begin{aligned}
&\max \sum_{i=1}^{n}\sum_{k=1}^{n} d_{ik}y_{ik} \\
&\sum_{k=1}^{n} y_{ik} = 1 \quad 1 \leq i \leq n \\
&\sum_{i=1}^{n} y_{ik} = 1 \quad 1 \leq k \leq n \\
&y_{ik} \in \{0,1\}
\end{aligned}$$

La única diferencia entre el problema QAP y el problema de asignación es la aparición de términos cuadráticos en la función objetivo, además del término lineal. Este problema es *NP-Hard*, y puede probarse (cfr. [Sah/76]) que la ϵ-aproximación de este problema también lo es. Los estudios poliedrales de este problema comenzaron en 1995 ([Rij/95], e independientemente, [Jun/96]), basados en una transformación de este problema a un problema de clique de peso máximo. En esta sección, se muestra cómo esta transformación se aplica al problema de *mapping* simple, y en la sección siguiente se transforma nuevamente el problema

[1] En [Jun/96] se presenta este problema como un problema de minimización. Utilizamos aquí una formulación equivalente como un problema de maximización, para simplificar la exposición.

a un problema de clique máxima (sin pesos), con el objetivo de formular un problema dual del problema inicial.

Veamos en primer lugar la transformación a un problema de clique de peso máximo, para adaptar luego esta transformación al problema de *mapping*. Sea $G_n = (V_n, E_n)$ el grafo con conjunto de nodos

$$V_n := \{(i,j) | 0 \leq i,j \leq n\}$$

y conjunto de ejes

$$E_n := \{\{(i,j),(k,l)\} | i \neq k, j \neq l, 0 \leq i,j,k,l \leq n\}.$$

la figura 4.1 muestra el grafo G_4, ubicando los nodos en forma de matriz. Como puede verse, el grafo G_n tiene todos los ejes posibles, excepto los ejes "horizontales" y "verticales". No es difícil verificar las siguientes propiedades de este grafo:

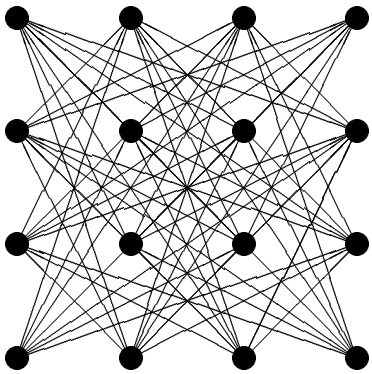


Figura 4.1: Grafo G_4.

Propiedad. *El tamaño de una clique máxima de G_n es n.*

Propiedad. *Una clique de G_n no tiene dos nodos en la misma fila o columna. Una clique máxima (es decir, de n nodos) contiene un nodo en cada fila y un nodo en cada columna.*

De esta forma, una clique máxima representa una asignación posible (con un nodo por fila y columna), y recíprocamente, cada asignación puede representarse por una clique. Si ahora asignamos pesos $h'_{ij} = h_{ij} + g_{ijij}$ al nodo (i,j) y pesos $g'_{ijkl} = g_{ijkl} + g_{klij}$ al eje $\{(i,j),(k,l)\}$, entonces el problema de asignación cuadrática es equivalente a encontrar la clique cuyos arcos y nodos tengan peso máximo entre todas las cliques de n nodos de G_n. Es decir, la función objetivo sobre una clique es la suma de los pesos de sus nodos y de sus arcos, y se busca la clique de n nodos con mayor función objetivo. Si CLQ_n denota al conjunto de cliques de n nodos de G_n, entonces QAP puede escribirse como:

$$\min \{h'(C) + g'(E_n(C)) : C \in CLQ_n\}.$$

Como se mencionó anteriormente, el problema de *mapping* es un caso particular del problema de asignación cuadrática, dado que podemos formular un modelo para el problema de *mapping* en términos de este último:

$$\begin{aligned}
\max \quad & \sum_{ij \in E(G)} \sum_{kl \in E(H)} y_{ik} y_{jl} \\
& \sum_{k \in V(H)} y_{ik} = 1 \quad \forall i \in V(G) \\
& \sum_{i \in V(G)} y_{ik} = 1 \quad \forall k \in V(H) \\
& y_{ik} \in \{0, 1\}
\end{aligned}$$

Las restricciones indican que cada tarea debe asignarse a un solo procesador, y que cada procesador debe recibir una sola tarea. La función objetivo cuenta la cantidad de ejes de G que se asignan a ejes de H: si $ij \in E(G)$, entonces $\sum_{kl \in E(H)} y_{ik} y_{jl} = 1$ si i y j se asignan en pocesadores k_0 y l_0 conectados (dado que $y_{ik_0} y_{jl_0} = 1$ y los restantes términos son nulos).

La transformación de QAP en un problema de clique máxima se puede aplicar para este problema, y se obtendrá una estructura particular, dado que se realiza a partir de un caso particular del problema general. Para realizar esta transformación, se considera la formulación anterior, pero multiplicando la función objetivo por 1/2. Esto no modifica el problema, y simplifica la transformación. Los coeficientes particulares de la función objetivo determinan los pesos asignados a los ejes y a los nodos.

Pesos de los nodos. Como la función objetivo del problema de *mapping* no tiene término lineal, entonces $h = 0$, y además, no está presente el término $y_{ik} y_{ik}$, con lo cual $g_{ikik} = 0$, y entonces $h'_{ik} = 0$. Es decir, los nodos de G_n tendrán peso 0.

Pesos de los ejes. En la función objetivo están presentes los términos $\frac{1}{2} y_{ik} y_{jl}$ con $ij \in E(G)$ y $kl \in E(H)$. Entonces, $g'_{ikjl} = g_{ikjl} + g_{jlik} = 1$ si $ij \in E(G)$ y $kl \in E(H)$, y $g'_{ikjl} = 0$ en caso contrario. Luego, el eje $\{(i,k),(j,l)\}$ tendrá peso 1 si ij y kl son ejes de G y H respectivamente, y tendrá peso nulo en caso contrario.

Llamemos $P(G,H)$ al grafo G_n obtenido por este procedimiento, junto con los pesos asociados a sus ejes. Con esto, se tiene que el problema de *mapping* es equivalente a encontrar la clique de peso máximo entre todas las cliques de tamaño n en $P(G,H)$. En este grafo, los nodos no tienen peso (tienen peso 0), y algunos ejes tienen peso 1 (los restantes tienen peso 0). En la próxima sección, se muestra cómo se puede transformar este problema en un problema de clique máxima (sin pesos), lo cual permite hallar un dual del problema original. Debe notarse que el grafo G_n es mucho más grande que los grafos del problema inicial, dado que tiene n^2 nodos. Como se mencionó anteriormente, esto hará que el problema dual hallado tenga grandes dimensiones, y sólo sea aplicable a ejemplos pequeños.

4.2 Transformación al problema de clique máxima

En la sección anterior, se mostró cómo expresar el problema original en términos de un problema de clique máxima sobre un grafo $P(G,H)$ con pesos 0-1 en sus ejes. En este grafo, el tamaño máximo de una clique es de n nodos, y se busca aquella con mayor peso en sus ejes. Mediante la transformación que se describe en esta sección, se puede llevar este grafo a un nuevo grafo $T(G,H)$, sin pesos en sus ejes, cuya clique máxima se corresponde con la clique de peso máximo en $P(G,H)$. Se describe primero esta transformación, para probar luego esta última propiedad.

Construcción de $T(G,H)$

Consideremos los ejes de peso 1 de $P(G,H)$. El nuevo grafo $T(G,H)$ se construye agregando nodos r_{ij} por cada eje ij de peso 1 de $P(G,H)$. Para simplificar la presentación, supongamos que los nodos originales de $P(G,H)$ son "blancos" y que los nuevos nodos son "rojos". Los nuevos nodos se conectan al resto del grafo agregando los siguientes ejes (ver fig. 4.2):

1. Se agregan ejes (i, r_{ij}) y (j, r_{ij}) entre el nuevo nodo y los extremos del eje que lo originó.
2. Se agregan todos los ejes (r_{ij}, t), para $t \in N(i) \cap N(j)$.
3. Se agregan todos los ejes $(r_{ij}, r_{i'j'})$, si $i', j' \in N(i) \cap N(j)$.

El nuevo grafo construido tiene la propiedad de preservar las cliques del grafo original, a las cuales se pueden agregar tantos nodos rojos como ejes de peso 1 del grafo original.

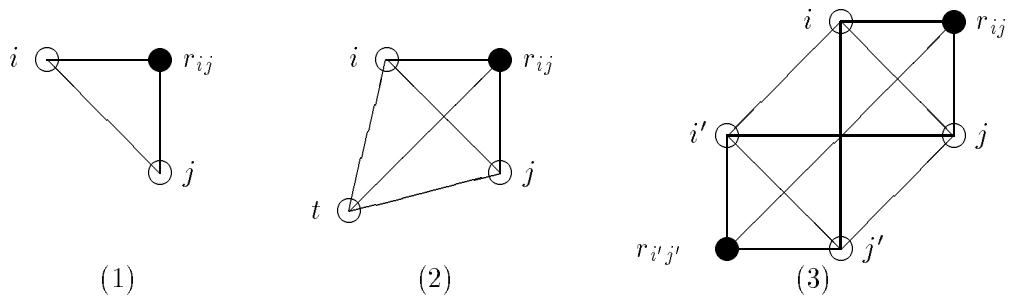


Figura 4.2: Ejes que se agregan en la transformación.

Lema. *Si existe una clique de peso k en $P(G,H)$, entonces existe una clique con k nodos rojos en $T(G,H)$.*

Demostración. Sea K una clique de peso k en $P(G,H)$. Los mismos nodos que forman esta clique en $P(G,H)$ forman una clique (de nodos blancos) en $T(G,H)$. Además, como la clique tiene peso k, entonces se encuentran en ella k ejes $e_1, ..., e_k$ de peso 1. Sea $e_s = ij$ uno de estos ejes. Como i y j están en la clique, $it \in E(P)$ y $jt \in E(P)$ $(\forall t \in K \backslash \{i,j\})$, y entonces

$(r_{ij}, t) \in E(T)$. Por otra parte, no es difícil verificar que todos los nodos rojos que se agregan a la clique están conectados entre sí, dado que sus nodos blancos correspondientes también lo están. Con esto, los nodos rojos asociados a los ejes $e_1, ..., e_k$ pueden agregarse a la clique en $T(G, H)$, con lo cual se tiene una clique en $T(G, H)$ con k nodos rojos.

Lema. *Si existe una clique con k nodos rojos en $T(G, H)$, entonces existe una clique de peso al menos k en $P(G, H)$.*

Demostración. Sea K una clique en $T(G, H)$ con k nodos rojos. Sea $r_{ij} \in K$ uno de estos nodos rojos, y supongamos que este nodo rojo está generado por el eje ij de peso 1 en $P(G, H)$. Como r_{ij} está en la clique, entonces $(r_{ij}, t) \in E(T)$ para cualquier otro nodo t de K.

1. Si t es un nodo blanco, r_{ij} está unido a t sólo si i y j son adyacentes a t.
2. Si t es un nodo rojo, r_{ij} es adyacente a t si los extremos i' y j' del eje que origina a t son ambos adyacentes a i y j, con lo cual i y j son también adyacentes a t.

En ambos casos, suponiendo que r_{ij} es adyacente a t se llega a que también i y j son adyacentes a t, con lo cual pueden agregarse a la clique K, manteniendo una clique (dado que están conectados con todos los nodos de la misma). De esta forma, se pueden agregar a la clique todos aquellos nodos blancos que sean extremos de los ejes que originan a los nodos rojos de K, obteniendo una clique compuesta por nodos blancos y rojos, de modo tal que *todos los nodos rojos son originados por algún par de nodos blancos de la clique.* Sea $K' = \{v \in K : v \text{ es blanco } \}$. K' es una clique de $P(G, H)$, y contiene al menos k ejes de peso 1, dado que origina en $T(G, H)$ esta cantidad de nodos rojos.

Teorema. *Si la clique de peso máximo en $P(G, H)$ tiene peso k, entonces la clique máxima en $T(G, H)$ tiene $n + k$ nodos.*

Demostración. Sea k el peso de la clique máxima en $P(G, H)$. Entonces, existe una clique $K \subseteq T(G, H)$ con k nodos rojos. Esta clique tiene un cierto número de nodos blancos, pero de acuerdo con la construcción de $P(G, H)$, siempre se puede extender la clique a una nueva clique de n nodos, y como los nodos blancos de $T(G, H)$ tienen la misma estructura que $P(G, H)$, se pueden agregar nodos blancos a K para obtener una nueva clique $K' \subseteq T(G, H)$ con n nodos blancos y k nodos rojos. Por último, niguna otra clique de $T(G, H)$ puede tener más de $n+k$ nodos, dado que no puede tener más de n nodos blancos (cualquier subconjunto de $n + 1$ nodos blancos no forma una clique, por la definición de $P(G, H)$) y no puede tener más de k nodos rojos, dado que en ese caso debería existir una clique en $P(G, H)$ con peso mayor que k, lo cual no puede suceder.

Observación. De acuerdo con la construcción del grafo $T(G, H)$, este grafo está formado por nodos blancos y rojos. Sin embargo, en las demostraciones de los lemas anteriores se prueba que una clique de k nodos rojos siempre puede extenderse a una clique de $n+k$ nodos blancos y rojos. Con esto, no es necesario que los nodos blancos permanezcan en este grafo. Se definió así para simplificar las demostraciones, pero de ahora en adelante, suponemos que el grafo $T(G, H)$ está formado solamente por nodos rojos.

Esta transformación permite obtener, entonces, un problema equivalente al problema original, dado que el valor óptimo del problema inicial está dado por la cantidad de nodos rojos en la clique máxima del grafo transformado $T(G,H)$. Este nuevo problema, equivalente al inicial, consiste en hallar la clique máxima de $T(G,H)$. Sea $\omega(G)$ el cardinal de la clique máxima de G, y sea $\kappa(G)$ su número cromático (cardinal del coloreo mínimo). Como $\omega(G) \leq \kappa(G)$, entonces el problema de coloreo mínimo de $T(G,H)$ es un problema dual del problema inicial, dado que cualquier coloreo tiene cardinal mayor o igual que la clique máxima de $T(G,H)$. Con esto, se tiene un dual combinatorio del problema inicial:

Problema dual. Dado el grafo de tareas G y el grafo de procesadores H, el problema consiste en hallar el coloreo mínimo de $T(G,H)$.

Este dual es, en principio, un dual débil, dado que existen grafos G para los cuales $\omega(G) < \kappa(G)$. El dual sería fuerte si $\omega(T) = \kappa(T)$, es decir si $T(G,H)$ fuera perfecto. En la sección siguiente, se muestra que el grafo $P(G,H)$ es un grafo perfecto, y se prueba que existen grafos G y H tales que $T(G,H)$ es perfecto. Esta es la mejor situación posible, dado que en este caso el dual obtenido es un dual fuerte.

4.3 Propiedades de $P(G,H)$ y $T(G,H)$

La estructura particular de $P(G,H)$ permite probar que se trata de un grafo perfecto (es decir, el cardinal de su clique máxima es igual a la cantidad de colores del coloreo mínimo). Un grafo es perfecto si y sólo si su complemento es perfecto (cfr. [Ber/85]).

Proposición. *El grafo $\bar{P}$ es perfecto.*

Demostración. El complemento de $P(G,H)$ está formado por el mismo conjunto de vértices, que se pueden ubicar sobre un arreglo rectangular. En este grafo, dos vértices están unidos por un eje si están sobre la misma fila o columna, dado que en $P(G,H)$ dos nodos son adyacentes si no están en la misma fila y no están en la misma columna.

Sea $A \subseteq V(P)$. El grafo $\bar{P}_A$ contiene los vértices de A y los ejes de $\bar{P}$ que unen puntos de A, es decir, aquellos que se encuentren sobre la misma fila o columna. Cualquier conjunto independiente de $\bar{P}_A$ estará formado por nodos sin filas ni columnas en común, dado que dos nodos en una misma fila o columna están unidos por un eje. Sea m_f el número de filas que tienen algún representante en A (es decir, la cantidad de filas cuya intersección con A es no vacía), y sea m_c el número de columnas con algún representante en A. Entonces, $\alpha(\bar{P}_A) = min\{m_f, m_c\}$, dado que cualquier conjunto con dos nodos sobre una misma fila o columna no será un conjunto independiente.

Por otra parte, todos los elementos sobre una misma fila o columna forman una clique, dado que cualquier par de nodos en una misma fila o columna es adyacente. Supongamos que $m_f < m_c$. Entonces, cada uno de los m_f subconjuntos de elementos en una misma fila forma una clique, con lo cual se tiene un recubrimiento por cliques de cardinal n_f. Sucede lo mismo si $m_c \geq m_f$, con lo cual $\theta(\bar{P}) \leq min\{m_f, m_c\}$. Como $\alpha(\bar{P}_A) \leq \theta(\bar{P}_A)$, entonces

Figura 4.3: Una instancia que genera un grafo perfecto

$\alpha(\bar{P}_A) = \theta(\bar{P}_A)$. Por lo tanto, $\bar{P}$ es perfecto.

Corolario. *El grafo $P(G,H)$ es perfecto.*

El grafo $P(G,H)$ es perfecto, pero el dual está definido sobre el grafo $T(G,H)$, que se obtiene mediante una transformación del primero. Es un problema abierto saber si este nuevo grafo es perfecto también, es decir, saber si la transformación produce nuevamente un grafo perfecto. Esta propiedad sería de gran importancia, dado que si $T(G,H)$ fuera perfecto, el dual obtenido sería un problema dual fuerte. El siguiente teorema muestra que esto efectivamente sucede en algunos casos.

Teorema. *Si el óptimo del problema de mapping es $|E(G)|$ (es decir, si todos los ejes del grafo de tareas se ubican en ejes del grafo de procesadores), entonces $T(G,H)$ es perfecto.*

Demostración. Como el óptimo del problema es $m = |E(G)|$, entonces la clique máxima de $T(G,H)$ tiene tamaño m, con lo cual $\omega(T) = m$. Para probar que este grafo es perfecto, hallamos un coloreo de sus nodos con m colores. Un coloreo está determinado por clases $K_1, ..., K_m$ de nodos, de modo tal que cada K_i induce un conjunto independiente. Supongamos que $E(G) = \{e_1, ..., e_m\}$. Podemos asignar a la clase K_i todos los nodos rojos que representan asignaciones del eje e_i (es decir, si $e_i = i_0 j_0$, entonces $K_i = \{(i_0 k, k_0 l) : kl \in E(H)\}$). Los nodos de K_i representan todas la asignaciones del eje e_i sobre los ejes de H, que son excluyentes entre sí, con lo cual K_i induce un conjunto independiente en $T(G,H)$, y puede colorearse con un mismo color. Con esto, se halló un coloreo de m nodos, y como $\omega(T) \leq \kappa(T)$, entonces $\omega(T) = \kappa(T)$. Por lo tanto, el grafo $T(G,H)$ es perfecto.

La construcción del coloreo de la demostración de este teorema hace pensar que $\kappa(T) = m$. Sin embargo, la instancia de la figura 4.3 origina un grafo $T(G,H)$ que es un circuito de 12 nodos (es decir, $T = C_{12}$), con lo cual $\omega(T) = \kappa(T) = 2$, a pesar de que $m = 4$. Este es un ejemplo de una instancia que tiene asociado un grafo $T(G,H)$ perfecto. Debe notarse que no se pudieron hallar instancias que definieran un grafo T que no fuese perfecto.

Significado de los ejes de $T(G,H)$. De acuerdo con la observación de la sección anterior, el grafo $T(G,H)$ está formado por nodos rojos, dado que se pueden eliminar los nodos blancos. La primera proposición de esta sección muestra el significado implícito que tienen los ejes entre nodos rojos. Un nodo rojo corresponde a un eje (ik, jl) de peso 1 de $P(G,H)$, que representa la asignación del eje $ij \in E(G)$ sobre el eje $kl \in E(H)$.

Lema. *Dos nodos rojos son adyacentes si y sólo si las asignaciones de ejes que representan no son excluyentes.*

Demostración. Dos nodos rojos son adyacentes si y sólo si los nodos blancos que los generan

son todos adyacentes entre sí, y esto sucede si y sólo si las asignaciones que ellos representan son posibles.

4.4 Experimentos computacionales

La construcción descripta permite transformar el problema original en un problema de clique máxima sobre un único grafo, con lo cual el problema de colorear este grafo es dual del problema inicial. Se realizaron algunos experimentos con este problema dual, que se resumen en esta sección. Estas pruebas permitieron conocer algunas características de los grafos involucrados en el problema dual, y confirmaron la dificultad de su resolución.

Observación. *El grafo $T(G,H)$ tiene $2m_G m_H$ nodos, siendo m_G y m_H la cantidad de ejes de G y H, respectivamente.*

Debe notarse que la transformación realizada permite reducir el problema inicial, cuyas instancias están compuestas por dos grafos, a un nuevo problema (clique máxima) con instancias de un solo grafo. El grafo que aparece en el problema dual "contiene" todas las posibles asignaciones y sus funciones objetivo, bajo la forma de cliques maximales. No es de extrañar, entonces, que este grafo tenga un gran tamaño, dado que "almacena" toda esta información.

La observación anterior permite intuir que la resolución del dual puede ser muy difícil, dado que si los grafos G y H tienen $O(n^2)$ ejes, entonces el grafo $T(G,H)$ del problema dual tendrá $O(n^4)$ **nodos**. Esto constituye una dificultad muy grande en términos de implementación, puesto que las cuestiones de almacenamiento se tornan importantes.

Tamaño del dual. Se realizaron pruebas sobre algunas instancias del problema, para conocer las características de los grafos que tienen asociados en el problema dual. La tabla 4.2 (ubicada al final del capítulo) muestra los datos obtenidos para un subconjunto representativo de instancias. La segunda columna contiene la cantidad de nodos del problema inicial, mientras que las columnas $v(T)$ y $e(T)$ muestran la cantidad de nodos y ejes del grafo del problema dual. Las últimas dos columnas contienen el grado mínimo $\delta(T)$ y máximo $\Delta(T)$ de este grafo, respectivamente.

Como se observó previamente, los grafos asociados al problema dual tienen gran tamaño, y las pruebas realizadas lo confirman (se obtuvieron grafos de hasta 19.000 nodos y 140.000.000 de ejes!). Una característica importante de estos grafos es su gran densidad, lo cual surgió cuando se realizaron estas pruebas y puede verificarse en la tabla 4.2. Al tener alta densidad, se torna muy difícil su coloreo mediante técnicas heurísticas (muchas de las heurísticas para coloreo de grafos están diseñadas únicamente para grafos poco densos). Con esto, se tienen dos razones fundamentales que cuestionan la utilidad práctica de este problema dual: su gran tamaño y la densidad de los grafos que origina.

Densidad del grafo $T(G,H)$. Puede realizarse una segunda observación con respecto a la densidad de estos grafos: La tabla 4.2 está ordenada por tamaño del problema original (cantidad de tareas), y puede verse que *la densidad de los grafos del dual aumenta junto*

con este tamaño. Es decir, cuanto más grande es el problema original, mayor es la densidad del grafo de su problema dual. Nuevamente, este hecho surgió cuando se realizaron los experimentos computacionales. Con el objetivo de constatar esta observación, se realizaron pruebas sobre un conjunto mayor de instancias, cuyos resultados se muestran en el gráfico 4.4. Este gráfico muestra la densidad del grafo dual vs. la cantidad de nodos del problema inicial, poniendo en evidencia una inesperada relación entre estos dos elementos.

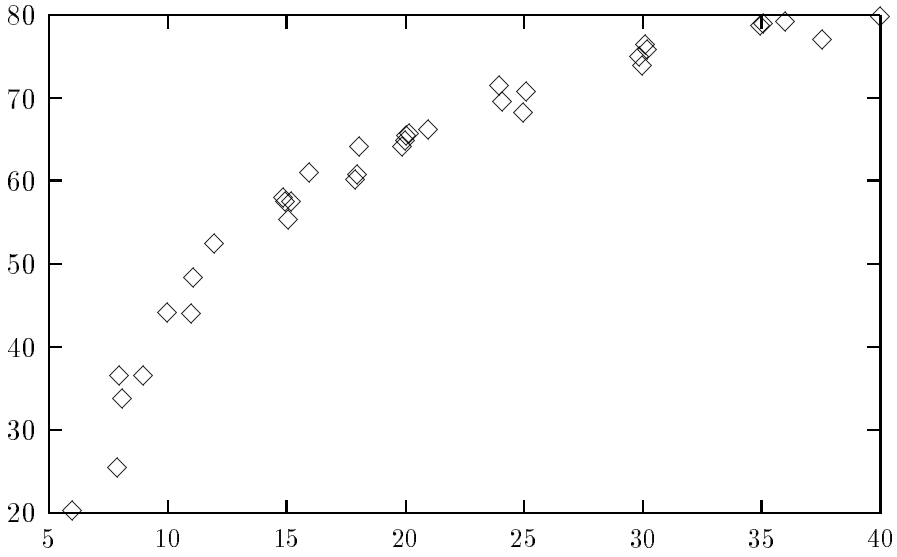


Figura 4.4: Densidad del grafo dual (porcentaje) en función de la cantidad de tareas de la instancia original

Esta relación, sin embargo, es empírica, y se observó cuando se resumieron los resultados. Puede hallarse una fórmula cerrada que indica la cantidad de ejes de T, pero se trata de una expresión complicada y muy difícil de manejar. Por estos motivos, no es inmediato deducir esta relación a partir de la expresión de la cantidad de ejes de este grafo.

Heurísticas de coloreo. Los experimentos descriptos muestran la enorme dificultad que presenta el problema dual. De todos modos, se realizaron pruebas con una heurística de coloreo para comprobar computacionalmente esta afirmación. Se utilizó la heurística para coloreo de grafos propuesta por Coll et al [Col/95], que combina las ideas de GRASP (cfr. [Feo/95]) con conceptos de búsqueda tabú (cfr. [Glo/95]), y que se describe en el anexo a este capítulo. Se usó al comienzo el código de los autores, pero el gran tamaño de algunas instancias provocó problemas de almacenamiento, con lo cual se reimplementó la heurística, generando el grafo "por demanda" en forma dinámica.

La tabla 4.1 resume los resultados obtenidos con la heurística de coloreo. Para cada problema, se muestran las cotas inferior y superior del óptimo obtenidas con el algoritmo *branch&cut*, para poder realizar una comparación con estos resultados. Las últimas dos columnas de la tabla indican la cota superior lograda por la heurística de coloreo sobre el problema dual y el tiempo utilizado. Se realizaron estas pruebas sobre una PC Pentium 233

Problema	Nodos	Cotas B&C	Cota Dual	Tiempo
STR1	9	14/14	35	14:22
STR2	6	8/8	8	47:29
DC1	18	14/14	27	17:26
DC2	15	14/14	62	2:54:03
PRB3	5	5/5	5	0:36
PRB4	5	5/5	5	2:13
PRB5	8	9/9	10	51:34
PRB6	8	10/10	10	1:08:58
PRB7	10	13/13	17	54:17
PRB7B	8	11/11	12	55:47
PRB8	11	13/13	25	20:36
PRB8B	8	6/6	6	1:25:10
PRB9	11	14/14	17	44:25
PRB9B	8	10/10	10	20:36
STR3	15	26/26	61	1:34:51
STR4	15	25/26	60	1:24:39
STR5	20	33/33	83	2:30:54
STR6	20	34/37	86	6:13:09
DF1	12	18/18	24	46:13
DF2	30	40/44	105	3:36:21
DF3	18	21/26	59	40:24
DF4	15	20/26	53	35:00
DF5	20	26/29	67	1:22:22
DF6	16	27/29	60	54:27
DF7	20	29/32	70	1:53:33

Tabla 4.1: Resultados de la heurística sobre el problema dual

Mhz, con 32 MB de memoria RAM.

Conclusión. En resumen, los resultados confirman la dificultad computacional del problema dual hallado, lo cual hace que este dual no tenga utilidad práctica. Sin embargo, este problema tiene interés teórico, dado que son pocos los problemas de optimización que tienen duales de natualeza combinatoria. Es posible que este dual sea débil, en cuyo caso el óptimo del dual puede no coincidir con el óptimo del problema inicial. Se probó que es fuerte sólo para algunos casos, aunque no se hallaron instancias en las cuales no lo fuera.

4.5 Anexo: Heurística de coloreo

En este anexo al presente capítulo se describe la heurística de coloreo utilizada en las pruebas computacionales sobre el problema dual. Esta heurística combina ideas de la técnica GRASP con conceptos propios de la búsqueda tabú, y ha arrojado buenos resultados al ser aplicada a problemas de tamaño mediano y grande (cfr. [Col/95]). Siguiendo las características prin-

cipales del método GRASP, la heurística consta de dos fases. En la primera fase se construye una solución con una componente aleatoria, que se intenta mejorar por medio de un procedimiento de búsqueda local en la segunda fase. Se repite este proceso una cantidad fija de veces, guardando la mejor solución hallada.

Fase de construcción. La primera fase construye un coloreo goloso, asignando en cada paso un color a algún nodo aún sin colorear. El nodo a colorear se elige en forma aleatoria de una lista de candidatos formada por todos los nodos cuya cantidad de adyacentes coloreados se encuentra en el intervalo $[(1-t)k, k]$, donde k es el número máximo de adyacentes coloreados entre todos los nodos aún sin colorear, y t es un parámetro del método (los autores sugieren tomar $t = 0.5$). El nodo elegido se colorea con el menor color posible, y se repite el procedimiento hasta colorear todos los nodos.

Fase de mejoramiento de la solución inicial. Se intenta mejorar la solución construida en la primera fase por medio de una búsqueda local híbrida, en la que participa una lista de colores tabú. Esta búsqueda intenta eliminar colores progresivamente de la solución actual, para obtener soluciones con menor cantidad de colores. Específicamente, en cada paso se elige un color c y se intenta cambiar el color de los nodos coloreados con c (sin aumentar la cantidad de colores de la solución). Si se pudo cambiar el color de todos estos nodos, se eliminó el color c. En caso contrario, este color ingresa a la lista tabú. La siguiente descripción especifica con mayor detalle la fase de mejora:

```
Comenzar con la lista tabu vacia.
Repetir h veces:

        Elegir un color c que no este en la lista tabu.

        Para cada nodo coloreado con c, intentar darle otro
        color, que sea usado en la solucion actual, priorizando
        los colores de la lista tabu.

        Si no se pudo eliminar el color c, agregarlo a la lista tabu.

Fin (repetir)
```

La lista tabú tiene un tamaño de $0.8r$ colores, siendo r la cantidad de colores utilizados en la solución actual. Cuando la lista está llena y un nuevo color ingresa a la lista, se elimina de ella el color más antiguo. El valor h indica la cantidad de intentos de eliminar colores en la solución y es un parámetro del método. En la implementación particular que se realizó, se detiene también el proceso si luego de una cantidad s de iteraciones no se logró mejorar la solución actual. Se utilizaron los parámetros $h = 100000$ y $s = 50$, repitiendo las dos fases un máximo de 150 veces.

Los autores recomiendan repetir pocas veces el proceso completo, utilizando en cambio valores de h grandes. Sin embargo, en las instancias del problema dual sobre las que se ejecutó este algoritmo, se obtuvieron mejores resultados con valores de h más pequeños (que

se reducen aún más al terminar el proceso luego de 50 iteraciones sin mejora), y repitiendo el proceso completo una mayor cantidad de veces. Se produce esta situación dado que la fase de mejoramiento logra reducir la cantidad de colores hasta un cierto punto, luego del cual la cantidad de colores de la solución permanece fija. En este momento, es conveniente recomenzar desde la primera fase, para intentar con otro coloreo inicial.

Problema	Nodos	$v(T)$	$e(T)$	Densidad	$\delta(T)$	$\Delta(T)$
STR2	6	240	5.760	20,08 %	44	52
PRB8B	8	200	5.040	25,32 %	32	100
PRB6	8	260	11.296	33,54 %	64	129
PRB5	8	200	7.240	36,38 %	58	100
STR1	9	1.008	184.464	36,34 %	336	406
PRB8	11	690	104.328	43,88 %	210	403
PRB7	10	448	44.036	43,97 %	159	245
PRB9	11	510	62.496	48,14 %	195	311
DF1	12	900	211.680	52,32 %	456	480
STR4	15	3.120	2.683.320	55,14 %	1.503	1.835
DC2	15	840	202.140	57,36 %	426	512
STR3	15	3.120	2.792.880	57,40 %	1.752	1.918
DF4	15	1.800	937.440	57,89 %	990	1.076
DC1	18	1.008	304.776	60,05 %	534	644
DF3	18	2.160	1.412.640	60,58 %	1.242	1.352
DF6	16	2.240	1.524.992	60,81 %	1.277	1.417
DF5	20	2.800	2.506.560	63,96 %	1.677	1.866
DF8	18	2.880	2.655.360	64,05 %	1.712	1.932
STR5	20	5.760	10.722.720	64,64 %	3.443	4.058
STR6	20	5.920	11.457.840	65,36 %	3.450	4.065
DF7	20	3.200	3.352.320	65,49 %	1.944	2.196
STR12	21	6.216	12.755.568	66,03 %	3.658	4.313
STR13	25	7.600	19.664.000	68,09 %	4.823	5.475
DF19	24	7.200	17.982.720	69,38 %	4.574	5.276
DF20	25	8.250	24.047.100	70,67 %	5.313	6.174
STR11	24	9.216	30.293.952	71,34 %	6.064	6.839
DF2	30	7.200	19.103.040	73,71 %	4.812	5.636
STR16	30	12.960	62.865.600	74,86 %	9.100	10.115
STR17	30	12.960	63.547.680	75,67 %	9.100	10.318
STR15	30	14.160	76.490.400	76,30 %	10.150	11.165
DC3	35	4.200	6.932.940	78,62 %	3.162	3.408
STR20	35	19.040	142.858.240	78,87 %	14.320	15.778
STR14	36	19.008	142.683.552	78,98 %	14.260	15.766
DC4	40	4.800	9.177.760	79,68 %	3.662	3.948

Tabla 4.2: Características del problema dual

Capítulo 5

Estudio poliedral del problema de mapping

"En la demostración de un teorema es indiferente el nombre de los puntos o segmentos, las letras latinas o griegas que los designan. No se demuestra una propiedad para un triángulo particular, sino para el triángulo en general; ni siquiera es necesario que esté bien dibujado y casi es mejor que no lo esté."

–Ernesto Sabato

El punto de partida de un algoritmo *branch&cut* es el estudio del poliedro asociado con el problema. Como se trata de un problema *NP-Hard*, entonces no es posible hallar una descripción completa de la cápsula convexa de los puntos definidos por el modelo de programación lineal entera, a menos que NP=coNP (cfr. [Nem/88]). Sin embargo, las desigualdades válidas que se encuentren pueden tener una contribución decisiva en un algoritmo *branch&cut*.

En este capítulo se describe el estudio del poliedro asociado al problema, y se presentan los resultados obtenidos. En primer lugar, se formulan dos modelos de programación lineal entera equivalentes para el problema y se presenta la dimensión del poliedro asociado con ellos. Luego, se enumeran las desigualdades válidas halladas, destacando los casos en los cuales estas desigualdades definen facetas.

5.1 Formulación del problema

Se presentan en esta sección dos formulaciones del problema, mediante modelos de programación entera. Se verá que estas formulaciones contienen las mismas variables y restricciones que el problema de asignación, junto con un grupo adicional de variables y restricciones que hacen que el problema resulte más complicado.

5.1.1 Primer modelo

En el modelo propuesto, se tienen dos grupos de variables. En primer lugar, se introducen las variables y_{ik}, con $i \in V(G), k \in V(H)$, que valen 1 si la tarea i se asigna al procesador k, y 0 en caso contrario. En segundo término, se tienen las variables c_{ij} ($ij \in E(G)$), que indican si el arco ij tiene sus extremos asignados a procesadores conectados (es decir, vale 1 si las tareas i y j se asignan a dos procesadores k y l tales que $kl \in E(H)$). Con estas variables, se puede formular un modelo para este problema [1].

$$
\begin{aligned}
\max \quad & \sum_{ij \in E(G)} c_{ij} \\
& \sum_{k \in V(H)} y_{ik} = 1 \qquad \forall i \in V(G) \\
& \sum_{i \in V(G)} y_{ik} = 1 \qquad \forall k \in V(H) \\
& c_{ij} \leq 2 - y_{ik} - y_{jl} \qquad \forall ij \in E(G), \forall kl \notin E(H) \\
& y_{ik} \in \{0,1\} \qquad \forall i \in V(G), \forall k \in V(H) \\
& c_{ij} \in \{0,1\} \qquad \forall ij \in E(G)
\end{aligned}
$$

En este modelo, se busca maximizar la cantidad de ejes de G que son "mapeados" a pares de procesadores adyacentes de H. Los dos primeros grupos de restricciones tienen la misma función que en el problema de asignación: cada tarea debe asignarse a un único procesador, y cada procesador debe recibir una sola tarea.

El tercer grupo de restricciones define los valores que pueden tomar las variables c_{ij}. Si las tareas i y j se encuentran asignadas a procesadores que no están conectados, por ejemplo k y l con $kl \notin E(H)$, entonces la restricción correspondiente a kl impone la condición:

$$c_{ij} \leq 2 - y_{ik} - y_{jl} = 2 - 1 - 1 = 0,$$

con lo cual $c_{ij} = 0$. Por otra parte, si las tareas i y j (con $ij \in E(G)$) se asignan a procesadores conectados, entonces ninguna de estas restricciones tendrá parte derecha nula (dado que se consideran los pares de procesadores no conectados, y estas dos tareas están asignadas a procesadores adyacentes). En este caso, las restricciones tendrán la forma $c_{ij} \leq 1$ o $c_{ij} \leq 2$, y como $c_{ij} \in \{0,1\}$, entonces se cumplirán trivialmente. Más aún, como se trata de un problema de maximización y esta variable aparece en la función objetivo con coeficiente positivo, entonces tendrá valor $c_{ij} = 1$ en el óptimo del problema.

Si llamamos $n = |V(G)| = |V(H)|$ a la cantidad de tareas, $m_G = |E(G)|$ a la cantidad de ejes del grafo de tareas y $\bar{m}_H = |\bar{E}(H)|$ a la cantidad de ejes del complemento de H, entonces puede verse que este modelo tiene $n^2 + m_G$ variables y $2n + m_G \bar{m}_H$ restricciones. Si el grafo de procesadores tiene $O(n)$ ejes (lo cual sucede en muchas de las instancias de este problema), entonces la cantidad de restricciones es $O(m_G n^2)$. El modelo que se presenta a continuación reduce esta cantidad a $O(m_G n)$ restricciones, manteniendo la misma cantidad de variables.

[1] Dado un nodo v, llamamos $N(v) := \{w \in V : vw \in E\}$ al conjunto de sus vecinos.

5.1.2 Segundo modelo

En este segundo modelo, se mantienen las mismas variables que en el modelo anterior, y sólo se modifican las restricciones que definen las variables de costo c_{ij}, reemplazándolas por una familia de restricciones equivalente (define los mismos puntos enteros), pero con menos restricciones. La segunda formulación es la siguiente:

$$
\begin{aligned}
\max \quad & \sum_{ij \in E(G)} c_{ij} \\
& \sum_{k \in V(H)} y_{ik} = 1 \qquad \forall i \in V(G) \\
& \sum_{i \in V(G)} y_{ik} = 1 \qquad \forall k \in V(H) \\
& c_{ij} + y_{ik} \leq 1 + \sum_{l \in N(k)} y_{jl} \qquad \forall ij \in E(G), \forall k \in V(H) \\
& y_{ik} \in \{0,1\} \qquad \forall i \in V(G), \forall k \in V(H) \\
& c_{ij} \in \{0,1\} \qquad \forall ij \in E(G)
\end{aligned}
$$

Sólo se modifican las restricciones que definen las variables de costo. Consideremos un eje ij del grafo de tareas y un procesador k. Si $y_{ik} = 0$, entonces la restricción definida por ij y k tiene la forma $c_{ij} \leq 1 + \sum y_{jl}$, que siempre es válida, pues $c_{ij} \leq 1$ y $\sum y_{jl} \geq 0$. Por otra parte, si $y_{ik} = 1$, entonces la tarea i está asignada al procesador k, y esta restricción tiene la forma

$$c_{ij} \leq \sum_{l \in N(k)} y_{jl},$$

con lo cual $c_{ij} \leq 1$ si la tarea j se asigna a algún vecino de k y $c_{ij} = 0$ en caso contrario. Nuevamente, la variable c_{ij} tiene libertad de situarse entre 0 y 1 si i y j se asignan a procesadores conectados. Como esta variable aparece en la función objetivo con coeficiente positivo, y las c_{ij} son independientes unas de otras, entonces en el óptimo todas las c_{ij} que puedan tomar el valor 1 efectivamente lo harán. Pueden agregarse restricciones que obliguen a que la variable c_{ij} tome el valor 1 cuando las tareas i y j se asignan a procesadores conectados, pero con esto se dificulta el análisis de la dimensión del poliedro, que pasa a depender de la estructura de los grafos del problema, y no fue posible hallar una expresión cerrada para la dimensión en este caso.

Al igual que el anterior, este nuevo modelo tiene $n^2 + m$ variables[2], pero la cantidad de restricciones de costo es mn, con lo cual se reduce en un orden de magnitud el tamaño del modelo anterior. Sin embargo, el hecho de tener un modelo más reducido no implica que los tiempos de resolución sean menores. Para comparar estos modelos, se realizaron pruebas con el algoritmo *branch&bound* provisto por CPLEX 4.0. La tabla 5.1 resume los resultados, mostrando para cada modelo las cotas inferior y superior alcanzadas (que coinciden si se

[2] De ahora en adelante, llamamos m a la cantidad m_G de ejes del grafo de tareas, para simplificar la notación.

resolvió el problema), la cantidad de nodos recorridos del árbol de enumeración, y el tiempo de resolución, con un límite de 1 hora (las instancias listadas se describen en la sección 7.1).

	Modelo 1				Modelo 2			
Problema	Inf.	Sup.	Nodos	Tiempo	Inf.	Sup.	Nodos	Tiempo
DC1	14	14	366	117.21 sg.	14	14	42	8.77 sg.
PRB5	9	9	1901	46.18 sg.	9	9	106	1.29 sg.
PRB6	10	10	6426	144.46 sg.	10	10	766	8.51 sg.
PRB7	12	14	38095	3600.00 sg.	13	13	11800	253.65 sg.
STR3	17	60	1670	3600.00 sg.	16	56	9605	3600.00 sg.
DC3	16	30	635	3600.00 sg.	27	30	2166	3600.00 sg.

Tabla 5.1: Resumen de las comparaciones entre los modelos

Como puede verse, en los casos que se pudieron resolver, el segundo modelo genera árboles más pequeños, al recorrer una menor cantidad de nodos. Este hecho, sumado al menor tamaño de los modelos lineales que se deben resolver, hace que el tiempo de resolución sea menor en estos casos. En los problemas que no se pudieron resolver, el menor tamaño de los subproblemas involucrados hace que el segundo modelo recorra una mayor cantidad de nodos en el mismo tiempo. Se observan resultados similares en otros problemas.

Observación. Debe notarse que ambos modelos son equivalentes, en el sentido de que definen exactamente las mismas soluciones factibles enteras. Por este motivo, al elegir el segundo modelo como base para el algoritmo *branch&cut*, las restricciones de costo del primer modelo se convierten en desigualdades válidas que se pueden incorporar al algoritmo.

5.2 Dimensión del poliedro

Los modelos presentados en la sección anterior definen un conjunto finito S de puntos en $\Re^{n^2+m}$ (soluciones factibles), y es de interés estudiar las propiedades de su cápsula convexa $conv(S)$, como se mencionó en la introducción, para identificar desigualdades válidas y analizar sus propiedades. La primera pregunta que surge al comenzar un estudio de este tipo se relaciona con la dimensión de $conv(S)$, que está dada por la cantidad máxima menos 1 de vectores afinmente independientes (afi) dentro de este conjunto[3]. Si se conoce la dimensión de este poliedro, entonces se tiene un dato muy importante para decidir si las desigualdades válidas halladas definen facetas de $conv(S)$.

En esta sección, se presenta la dimensión del poliedro definido por las restricciones del segundo modelo, utilizando para esto información conocida sobre el poliedro del problema de asignación (presentado en la sección 4.1.1), que está definido por las siguientes restricciones:

$$\sum_{k \in V(H)} y_{ik} = 1 \qquad \forall i \in V(G),$$

$$\sum_{i \in V(G)} y_{ik} = 1 \qquad \forall k \in V(H).$$

[3]Los vectores $x_1, ..., x_n$ son afi si la única solución α de $\sum \alpha_i x_i = 0$, $\sum \alpha_i = 0$ es $\alpha = 0$.

Llamemos T al conjunto de las soluciones factibles enteras del problema de asignación, y $conv(T)$ a su cápsula convexa. La matriz correspondiente a este problema es totalmente unimodular, con lo cual todos los extremos de la relajación lineal son enteros. Dada la similitud de este modelo con el del problema de *mapping*, podría pensarse que sucede lo mismo en este caso. Sin embargo, esto no ocurre así.

Proposición. $dim(conv(T)) = (n-1)^2$.

Demostración. La matriz definida por las restricciones de asignación tiene rango $2n-1$, y como esta matriz es totalmente unimodular, entonces todos los extremos de la relajación lineal son enteros (cfr. [Nem/88]). Con esto, se tiene que la relajación lineal coincide con la cápsula convexa del poliedro de asignación. Como la matriz tiene rango $2n-1$ y tiene n^2 variables, entonces la dimensión del poliedro de asignación es de $n^2 - rg(A) = n^2 - 2n + 1 = (n-1)^2$. Luego, la cantidad máxima de vectores afi en el poliedro de asignación es de $(n-1)^2 + 1$.

El siguiente teorema presenta la dimensión del poliedro definido por la formulación anterior. Sea S el conjunto de puntos que cumplen las restricciones del modelo, y sea $conv(S)$ su cápsula convexa. Llamemos también n a la cantidad de tareas ($n = |V(G)| = |V(H)|$) y m a la cantidad de ejes de G (es decir, $m = |E(G)|$). Se asume además que $E(H) \neq \emptyset$ (al menos un par de procesadores está conectado).

Teorema. $dim(conv(S)) = (n-1)^2 + m$.

Demostración. Veamos primero que $dim(conv(S)) \geq (n-1)^2 + m$, para lo cual se buscan puntos afi dentro del poliedro. De acuerdo con la proposición anterior, tenemos $(n-1)^2 + 1$ vectores afi que cumplen las restricciones de asignación. Podemos extender estos vectores al espacio del problema de *mapping*, ubicando ceros en las posiciones correspondientes a las variables c_{ij}. Estos nuevos vectores cumplen todas las restricciones del problema de *mapping*, y son $(n-1)^2 + 1$ vectores afi.

Para obtener los restantes m vectores afi, consideremos las variables c_{ij}. Podemos hallar un vector x^{ij} por cada una de estas variables, de modo tal que $x^{ij}_{ij} = 1$ y $x^{ij}_{rs} = 0$ para $rs \neq ij$ (es decir, la variable c_{ij} vale 1, y las restantes c_{rs} son nulas). El vector x^{ij} es el vector característico de la siguiente asignación: se ubican las tareas i y j en procesadores vecinos (que existen por hipótesis), y las restantes tareas en cualquier procesador disponible, fijando $c_{ij} = 1$ y las restantes variables c_{rs} en cero (el modelo lo permite). Cada uno de estos vectores x^{ij} es li con los vectores hallados anteriormente, pues en estos vectores todas las variables c_{ij} son nulas. Entonces, estos nuevos vectores son afi con los anteriores, y hemos hallado $(n-1)^2 + m + 1$ vectores afi dentro del poliedro, con lo cual $dim(conv(S)) \geq (n-1)^2 + m$.

Veamos ahora que la dimensión no puede ser mayor que $(n-1)^2 + m$. No es difícil verificar que las $2n$ igualdades del modelo forman una matriz de rango $2n-1$. Si quitamos una de ellas, se obtiene un sistema de ecuaciones linealmente independientes cumplido por todos los puntos factibles (aunque puede no ser completo). Como se tienen $n^2 + m$ variables, entonces

$$dim(conv(S)) \leq n^2 + m - (2n-1) = (n-1)^2 + m.$$

Por lo tanto, $dim(conv(S)) = (n-1)^2 + m$.

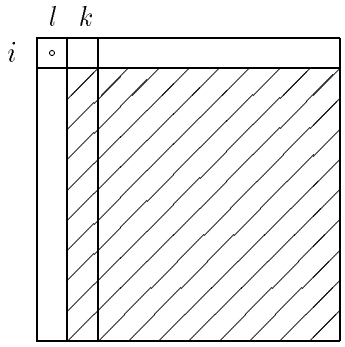
Si $\pi x \leq \pi_0$ es una desigualdad válida (es decir, es cumplida por todos los puntos de S), entonces $F = \{x : \pi x = \pi_0\} \cap conv(S)$ es una *cara* de $conv(S)$ si F es no vacío. Una *faceta* de $conv(S)$ es una cara maximal, con relación a la inclusión. Conociendo la dimensión de la cápsula convexa de los vectores característicos de las soluciones del problema, se tiene un dato muy importante para decidir si las desigualdades halladas definen facetas de $conv(S)$. Esta propiedad es de interés, dado que esta es la mejor situación posible desde el punto de vistateórico, al ser las facetas necesarias y suficientes para definir $conv(S)$.

Cuando se encuentra una desigualdad válida para S, se trata de decidir si esta desigualdad define una faceta de $conv(S)$, lo cual proporciona una garantía de su calidad teórica, al no estar dominada por ninguna otra desigualdad válida. Existen resultados experimentales que muestran que las clases de facetas (o desigualdades que definen caras de dimensión alta, aunque no sean facetas) pueden ser extremadamente útiles en el contexto de un algoritmo de planos de corte o de tipo *branch&cut*.

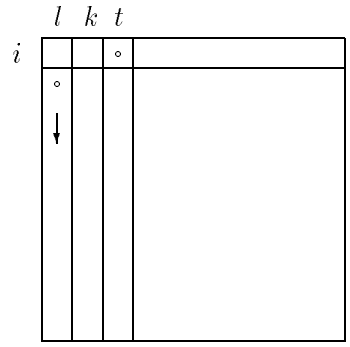
El hecho de que la cara F definida por una desigualdad válida constituya una faceta de $conv(S)$ está relacionado con la dimesión de F. Puede probarse (cfr. [Nem/88]) que F es faceta si y sólo si $dim(F) = dim(conv(S)) - 1$. Este resultado muestra la importancia de conocer la dimensión de $conv(S)$, y se utiliza con frecuencia a lo largo del capítulo. En este sentido, puede verse que las restricciones de no negatividad $y_{ik} \geq 0$ definen facetas de la cápsula convexa.

Proposición. *La desigualdad $y_{ik} \geq 0$ define una faceta de $conv(S)$.*

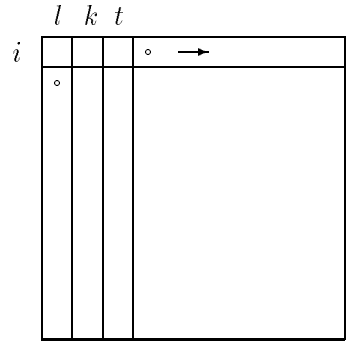
Demostración. En primer lugar, puede verse que esta desigualdad define una cara de $conv(S)$, dado que es desigualdad válida para S y además $S \cap \{(y,c) : y_{ik} = 0\} \neq \emptyset$. Para ver que define una faceta, se deben hallar $dim(conv(S))$ vectores afi que cumplan esta restricción por igualdad (es decir, vectores $(y,c) \in S$ tales que $y_{ik} = 0$). Se presentan estos vectores en tres grupos. Sean $l, t \in V(H) \backslash k$ $(l \neq t)$.



En primer lugar, fijemos $y_{il} = 1$. Esto fuerza a que $y_{jl} = y_{ip} = 0$, $\forall j \neq i$, $\forall p \neq l$ (en particular, $y_{ik} = 0$, con lo cual se cumple la restricción por igualdad). Si borramos todas estas variables, tenemos un problema de transporte de $n-1$ nodos, con lo cual podemos encontrar $(n-2)^2 + 1$ vectores afi para este nuevo problema. Extendiendolos de modo tal que $y_{il} = y_{ip} = c_{ij} = 0$, tenemos vectores de $conv(S)$ que son afi y cumplen $y_{ik} \geq 0$ por igualdad.



En segundo lugar, fijemos $y_{it} = 1$ y consideremos los $n-1$ vectores que se obtienen fijando $y_{jl} = 1$ para los distintos valores de $j \neq i$ (las restantes variables se fijan arbitrariamente, de modo tal de respetar las restricciones de transporte). Cada nuevo vector de este grupo es li con los anteriores (y con los del primer grupo), dado que los anteriores tienen $y_{jl} = 0$, y cumplen $y_{ik} = 0$.



Por último, consideremos vectores obtenidos al fijar $y_{ip} = 1$, con $p \neq k, l, t$. Nuevamente, las restantes tareas se asignan a cualquier procesador, respetando las restricciones de transporte (siempre se puede hacer). Se obtienen $n-3$ vectores que son li con los anteriores, dado que $y_{ip} = 0$ en los grupos precedentes (y en los vectores precedentes de este mismo grupo). Además, cumplen $y_{ik} = 0$, pues $y_{ip} = 1$ y $p \neq i$.

En los vectores de los tres grupos anteriores, se fijan las variables c_{ij} en cero, lo cual siempre se puede hacer. Se obtuvieron, entonces, $[(n-2)^2+1]+[n-1]+[n-3] = n^2-2n+1 = (n-1)^2$ vectores afi, que están dentro de $S \cap \{(y,c) : y_{ij} = 0\}$, y que tienen las variables c_{ij} en cero.

Para completar la demostración, se deben encontrar otros m vectores afi. Para cada $ij \in E(G)$, consideremos el vector característico obtenido al asignar las tareas i y j a procesadores adyacentes, y las restantes tareas en cualquier procesador, de modo arbitrario. Fijamos además, $c_{ij} = 1$ y las restantes variables $c_{rs} = 0$. Con esto, obtenemos m vectores li con los anteriores, con lo cual son afi.

Se han hallado, entonces, $(n-1)^2+m = dim(conv(S))$ vectores afi que cumplen $y_{ik} \geq 0$ por igualdad. Luego, la dimensión de la cara definida por esta desigualdad es $dim(conv(S)) - 1$, con lo cual $y_{ij} \geq 0$ define una faceta de $conv(S)$.

5.2.1 Análisis de las restricciones de costo

La única diferencia entre los dos modelos planteados para el problema consiste en el modo de relacionar las variables de costo c_{ij} con las variables de asignación y_{ik}. En el primero se tenían las $O(n^2m)$ restricciones

$$c_{ij} + y_{ik} + y_{jl} \leq 2, \tag{5.1}$$

para $kl \notin E(H)$, mientras que en el segundo modelo se tienen la siguiente familia de $O(nm)$ restricciones:

$$c_{ij} + y_{ik} \leq 1 + \sum_{l \in N(k)} y_{jl} \tag{5.2}$$

En esta sección, se muestra que las restricciones 5.2 se pueden obtener a partir de 5.1 por medio del procedimiento de Chvatal-Gomory, y se prueba que estas restricciones definen facetas de $conv(S)$. Por otra parte, se muestra un caso particular en el cual 5.1 también definen facetas de este poliedro.

Veamos en primer lugar cómo se pueden obtener las restricciones 5.2 a partir del primer modelo, mediante el procedimiento de Chvatal-Gomory, que se define del siguiente modo: Si $Ax \leq b$ es el modelo original y $u \in Z^m$, entonces $uAx \leq \lfloor ub \rfloor$ es una CG-desigualdad. Se puede probar que todas las desigualdades válidas posibles pueden generarse o están dominadas por las CG-desigualdades (cfr. [Nem/88]).

Consideremos un eje $ij \in E(G)$ y un procesador $k \in V(H)$. Sea $\bar{N}(k) = V \backslash N(k)$, y $\bar{d}(k) = |\bar{N}(k)|$. Para cada procesador $l \in \bar{N}(k)$ se tiene la restricción 5.1. Llamemos $d = \bar{d}(k)$. En primer lugar, la suma de todas estas restricciones es:

$$dc_{ij} \leq 2d - dy_{ik} - \sum_{l \in \bar{N}(k)} y_{jl}.$$

Además, si restamos de $\sum_l y_{jl} = 1$ las restricciones $y_{jl} \geq 0$ para $l \in N(k)$, se tiene que $\sum_{l \in \bar{N}(k)} y_{jl} \leq 1$, con lo cual $(d-1) \sum y_{jl} \leq d - 1$. Sumando esta condición a la desigualdad anterior, se tiene que:

$$dc_{ij} + d \sum y_{jl} \leq 2d - dy_{ik} + d - 1$$

$$d(c_{ij} + \sum y_{jl} + y_{ik}) \leq 3d - 1$$

$$c_{ij} + \sum y_{jl} + y_{ik} \leq (3d-1)/d$$

Como la porción izquierda de la desigualdad anterior es entera, entonces se puede tomar la parte entera de $(3d-1)/d$, y la desigualdad se sigue cumpliendo. Además, $\lfloor (3d-1)/d \rfloor = \lfloor 3 - 1/d \rfloor = 2$, con lo cual se ha probado la siguiente proposición.

Proposición. *Si $ij \in E(G)$ y $k \in V(H)$, entonces*

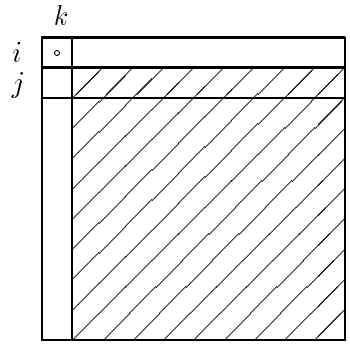
$$c_{ij} \leq 2 - y_{ik} - \sum_{l \in \bar{N}(k)} y_{jl} \tag{5.3}$$

es desigualdad válida para $conv(S)$. Además, esta desigualdad es C-G de rango 1 con respecto a las restricciones 5.1.

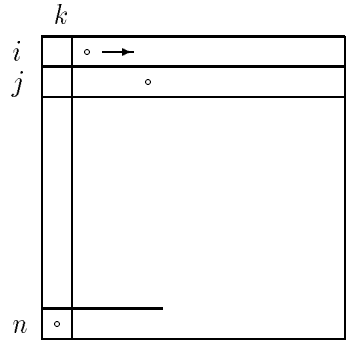
No es difícil verificar que 5.2 y 5.3 son en realidad la misma desigualdad, dado que $\sum_{l \in N(k)} y_{jl} = 1 - \sum_{l \in \bar{N}(k)} y_{jl}$. Veamos ahora que estas desigualdades definen facetas de $conv(S)$, bajo condiciones generales.

Teorema. *La desigualdad 5.3 define una faceta de* $conv(S)$ *si* $N(l)\backslash(N(k)\cup\{k\}) \neq \emptyset$, *para todo procesador* $l \neq k$.

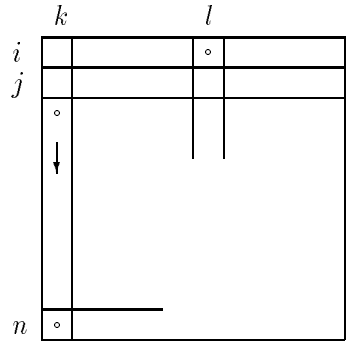
Demostración. Nuevamente, se construyen $(n-1)^2+m$ puntos afi dentro de S y que cumplen 5.3 por igualdad, con lo cual se prueba que esta desigualdad define una faceta.



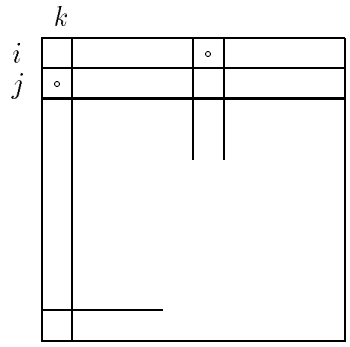
Fijando la tarea i en el procesador k, se tiene un problema de asignación reducido con $n-1$ tareas y procesadores (indicado por el rayado diagonal), para el cual pueden hallarse $(n-2)^2+1$ vectores afi. Extendemos estos vectores fijando las variables de costo en cero, excepto c_{ij} que se fija en 1 si la tarea j se asigna a un vecino de k.



El segundo grupo se obtiene asignando la tarea i a cada procesador $l \neq k$, y la tarea j sobre algún procesador en $N(l)\backslash(N(k)\cup\{k\})$ (este procesador existe por hipótesis). El procesador k recibe la tarea n (asumimos que $i \neq n$ y $j \neq n$), y las restantes tareas se asignan arbitrariamente. Nuevamente, se fijan las variables de costo en cero, excepto c_{ij}, que se ajusta en 1, para cumplir 5.3 por igualdad, dado que i y j se asignaron en procesadores conectados[4].



Sea $l \neq k$. Asignando la tarea i al procesador l y la tarea j a un procesador de $N(l)\backslash N(k)$, se puede fijar $c_{ij}=1$, con lo cual se cumple la restricción por igualdad. El tercer grupo tiene $n-3$ vectores, y se obtiene asignando al procesador k las tareas $t \in V(G)\backslash\{i,j,n\}$. Cada nuevo vector de este grupo es afi con los anteriores, en los cuales la variable y_{tk} es nula.



El cuarto grupo está formado por un sólo vector, obtenido al asignar la tarea j al procesador k y la tarea i a un procesador de $N(k)$ (con lo cual puede fijarse $c_{ij} = 1$ y cumplir 5.3 por igualdad). En los vectores de los grupos anteriores, $y_{jk} = 0$, con lo cual este punto es afi con ellos.

Por último, se tienen $m - 1$ vectores, uno para cada eje $rs \neq ij$, en los cuales $c_{rs} = 1$ (se asignan las tareas para que esto sea posible) y las restantes variables de costo son nulas. Se tienen, por lo tanto, $(n-1)^2 + m$ vectores afi que cumplen 5.3 por igualdad, con lo cual esta desigualdad define una faceta de $conv(S)$.

El teorema anterior muestra que las restricciones de costo del segundo modelo definen facetas del poliedro del problema, siempre que $N(l) \backslash N(k) \neq \emptyset$. Estas condiciones son válidas para las arquitecturas más comunes (grillas con 4, 6 y 8 vecinos por nodo, hipercubos, etc.). Por su parte, una pequeña variación de las restricciones 5.1 de costo del primer modelo define una faceta pero en casos más restrictivos, de acuerdo con la siguiente proposición, que se demuestra de modo similar al teorema precedente.

Proposición. *Sean k y l dos procesadores no conectados, tales que $d(k) = d(l) = n - 2$ (es decir, son vecinos de todos los restantes procesadores). Entonces,*

$$c_{ij} + (y_{ik} + y_{jk}) + (y_{il} + y_{jl}) \leq 2$$

define una faceta de $conv(S)$.

En esta nueva desigualdad, se añadió el término simétrico $(y_{il} + y_{jk})$ a la restricción 5.1, que no modifica su validez pero logra una desigualdad de mayor fuerza.

Se presentan a continuación distintas familias de desigualdades válidas para S, realizando un análisis de su calidad cuando esto es posible, y puntualizando los casos en los cuales las desigualdades definen facetas. En la siguiente sección, se presenta una desigualdad que surge inmediatamente de la formulación del problema, sobre la cual se realizan distintas variaciones, obteniendo una familia de facetas del poliedro. En la sección 5.4, se presentan las familias exponenciales de desigualdades halladas, incluyendo tres familias de facetas con una cantidad exponencial de desigualdades. Por último, se incluyen otras familias polinomiales en la sección 5.5.

En muchos casos, se parte de desigualdades que son válidas solamente para algunos puntos de S, y se utiliza el procedimiento de *lifting* para obtener a partir de ellas nuevas desigualdades que sean válidas para todo el conjunto S de soluciones factibles. Más precisamente, si $S' \subseteq S$ es el conjunto de puntos de S en los cuales algunas de las variables están fijas, y si tenemos una desigualdad válida para S', se busca una nueva desigualdad válida para S en la cual no

se modifiquen los coeficientes que acompañan a las variables que no estaban fijas en S'. El procedimiento particular de *lifting* que se utilizó corresponde al descripto en [Nem/88], y se especifica a continuación, para referencia a lo largo del capítulo.

5.2.2 Procedimiento de *lifting*

Sea $B = \{0, 1\}$, y supongamos que $S \subseteq B^n$. Llamemos $S^0 := \{x \in S : x_1 = 0\}$ y $S^1 := \{x \in S : x_1 = 1\}$, de modo tal que $S = S^0 \cup S^1$. Si tenemos una desigualdad $\sum_{j=1}^{n} \pi_j x_j \leq \pi_0$ válida para S^0, el procedimiento de *lifting* permite obtener una nueva desigualdad $\alpha_1 x_1 + \sum_{j=1}^{n} \pi_j x_j \leq \pi_0$ válida para S.

Proposición. *Supongamos que*

$$\sum_{j=1}^{n} \pi_j x_j \leq \pi_0 \tag{5.4}$$

es válida para S^0. *Si* $S^1 = \emptyset$, *entonces* $x^1 \leq 0$ *es válida para* S. *Si* $S^1 \neq \emptyset$, *entonces*

$$\alpha_1 x_1 + \sum_{j=1}^{n} \pi_j x_j \leq \pi_0 \tag{5.5}$$

es válida para S *para cualquier* $\alpha_1 \leq \pi_0 - \zeta$, *donde* $\zeta = max\ \{\sum_{j=2}^{n} \pi_j x_j : x \in S^1\}$.

Demostración. Sea $\bar{x} \in S$. Si $\bar{x} \in S^0$, entonces

$$\alpha_1 \bar{x}_1 + \sum_{j=1}^{n} \pi_j \bar{x}_j = \sum_{j=1}^{n} \pi_j \bar{x}_j \leq \pi_0$$

es válida para S, dado que $x_0 = 0$ y 5.5 es válida para S^0. Por otra parte, si $\bar{x} \in S^1$, entonces

$$\alpha_1 \bar{x}_1 + \sum_{j=1}^{n} \pi_j \bar{x}_j = \alpha_1 + \sum_{j=1}^{n} \pi_j \bar{x}_j \leq \alpha_1 + \zeta \leq \pi_0,$$

por definición de α_1 y ζ.

Cuando $\alpha_1 = \pi_0 - \zeta$, el *lifting* se dice máximo. En este caso, la dimensión de la cara definida por 5.5 es mayor que la dimensión de la cara definida por 5.4.

Proposición. *Bajo las mismas condiciones que en la proposición anterior, si* $\alpha_1 = \pi_0 - \zeta$ *y 5.4 define una cara de dimensión* k *de* $conv(S^0)$, *entonces 5.5 define una cara de dimensión* $k + 1$ *de* $conv(S)$.

Demostración. Como 5.4 define una cara k-dimensional de $conv(S^0)$, existen puntos $x^i \in S^0$, para $i = 1, ..., k+1$ que son afi y cumplen 5.4 por igualdad. Como $x^i = 0$, entonces x^i cumple

5.5 por igualdad. Sea $x^* \in S^1$ tal que $\zeta = \sum_{j=2}^{n} \pi_j x_j^*$. Con $\alpha_1 = \pi_0 - \zeta$, este punto cumple 5.5 por igualdad, y como $x_1^* = 1$, entonces x^* no puede ser escrito como combinación afín de $x^1, ..., x^{k+1}$, con lo cual los $k+2$ vectores $x^*, x^1, ..., x^{k+1}$ son afi.

Corolario. *Si 5.4 define una faceta de $conv(S^0)$, entonces 5.5 define una faceta de $conv(S)$.*

El principio de *lifting* también puede aplicarse cuando la variable x_1 se encuentra fija en su cota superior.

Proposición. *Supongamos que 5.4 es válida para S^1. Si $S^0 =$, entonces $x^1 \geq 1$ es válida para S. Si $S^0 \neq$, entonces*

$$\gamma_1 x_1 + \sum_{j=1}^{n} \pi_j x_j \leq \pi_0 + \gamma_1 \tag{5.6}$$

es válida para S para cualquier $\gamma_1 \geq \zeta - \pi_0$, donde $\zeta = max\ \{\sum_{j=2}^{n} \pi_j x_j : x \in S^0\}$.

Del mismo modo que en el caso anterior, si 5.4 define una cara de dimensión k de $conv(S^1)$, entonces 5.6 define una cara de dimensión $k+1$ de $conv(S)$.

5.3 Variaciones sobre una desigualdad válida

En la siguiente proposición, se presenta una desigualdad sencilla que es válida para algunos puntos del problema. Recordemos que si v es un nodo, entonces $N(v)$ denota al conjunto de sus vecinos.

Proposición. *Si $i \in V(G)$ y $k \in V(H)$, entonces*

$$\sum_{j \in N(i)} c_{ij} \leq \sum_{j \in N(i)} \sum_{l \in N(k)} y_{jl} \tag{5.7}$$

es desigualdad válida para $S' := S \cap \{(y, c) : y_{ik} = 1\}$.

Demostración. Como $y_{ik} = 1$, entonces la tarea i está asignada al procesador k. Por otra parte, las variables c_{ij}, con $j \in N(i)$, indican la cantidad de vecinos de i que están asignados a procesadores vecinos de k (es decir, en $N(k)$), y que aportan a la función objetivo (un vecino j puede asignarse a $N(k)$ pero tener la variable c_{ij} en cero). Esta cantidad está acotada por $\sum_{j \in N(i)} \sum_{l \in N(k)} y_{jl}$ (cuando todas las variables c_{ij} que pueden valer 1 efectivamente lo hacen), con lo cual la condición 5.7 debe cumplirse por todos los puntos con $y_{ik} = 1$.

La desigualdad anterior es válida sólo para algunos puntos, pero puede modificarse de modo tal que sea válida para todo el conjunto S. Esta desigualdad se cumple cuando $y_{ik} = 1$, y

para hacer que se cumpla en todos los puntos de S, podemos añadir un término suficientemente grande, que se anule cuando $y_{ik} = 1$, logrando la siguiente desigualdad ($d(i) = |N(i)|$ denota el grado de la tarea i):

$$\sum_{j \in N(i)} c_{ij} \leq \sum_{j \in N(i)} \sum_{l \in N(k)} y_{jl} + d(i)(1 - y_{ik})$$

Cuando $y_{ik} = 1$, entonces $N(k)(1 - y_{ik}) = 0$, y la desigualdad es igual a la anterior. Por su parte, cuando $y_{ik} = 0$, esta nueva desigualdad se cumple trivialmente, dado que $\sum_{j \in N(i)} c_{ij} \leq d(i)$. Como la desigualdad se cumple en todos los puntos de S, entonces es una desigualdad válida para este poliedro.

***

La desigualdad 5.7 es válida para los puntos de S con $y_{ik} = 1$, y en el parágrafo anterior se obtuvo una desigualdad válida para todos los puntos de S añadiendo un término suficientemente grande, de modo tal que siguiera valiendo cuando $y_{ik} = 0$. Este procedimiento corresponde a un paso de *lifting* sobre la variable y_{ik}, que estaba fija en su cota superior. Sin embargo, el proceso de *lifting* puede aplicarse de otra forma, de modo tal de obtener una desigualdad (válida para todo el poliedro) que define una faceta de $conv(S)$.

Dada una tarea i, la desigualdad 5.7 considera todos sus vecinos (nodos de $N(i)$), y realiza una acotación en conjunto para todos ellos. Si en lugar de esto, consideramos un sólo vecino $j \in N(i)$ fijo, la desigualdad anterior se escribe como:

$$c_{ij} \leq \sum_{l \in N(k)} y_{jl} \tag{5.8}$$

La variable c_{ij} está forzada a valer 0 si j no se asigna sobre ninguno de los procesadores vecinos de k. Esta desigualdad es válida para $S' = S \cap \{(y, c) : y_{ik} = 1\}$, y la siguiente proposición muestra que define una faceta de $conv(S')$.

Proposición. *Sea F la cara definida por 5.8. Entonces F es una faceta de $conv(S')$.*

Demostración. El conjunto S' puede considerarse como un problema de *mapping* reducido, en el cual ya se asignó la tarea i al procesador k. La misma demostración para S se aplica a S', con lo cual no es difícil verificar que $dim(conv(S')) = (n-2)^2 + m$. Para probar la proposición, buscamos $(n-2)^2 + m$ vectores afi en F, que se dividen en dos grupos:

1. Las variables y_{jl} con $j \neq i$ o $l \neq k$ conforman un problema de asignación de $n-1$ tareas, con lo cual pueden hallarse $k = (n-2)^2 + 1$ vectores $y_1, y_2, ..., y_k \in \Re^{(n-1)^2}$ afi en este problema de asignación reducido. Podemos extender estos vectores a la dimensión del problema original, fijando $y_{ik} = 1$ y las restantes variables en cero, excepto c_{ij}, que se fija en 1 si $\sum_{l \in N(k)} y_{jk} = 1$. De esta forma, se tienen $(n-2)^2 + 1$ vectores afi dentro de F (dado que cumplen 5.8 por igualdad).

2. Ordenemos las variables de costo de modo tal que la primera sea c_{ij} (que participa en la desigualdad). El segundo grupo de vectores está formado por $m-1$ vectores $z_1, ..., z_m$ de modo tal que en z_t sólo la t-ésima variable de costo es no nula. Supongamos que la t-ésima variable de costo es c_{rs}. Elegimos entonces $z_t = (y^t, c^t)$ de modo tal que r y s se asignen a procesadores conectados (con lo cual c_{rs} puede fijarse en uno), y fijamos las restantes variables de costo en cero, excepto c_{ij}, que se fija en 1 si $\sum_{l \in N(k)} y_{jk} = 1$ (con lo cual se cumple 5.8 por igualdad). Se tienen, entonces, $m-1$ vectores afi entre sí y con los anteriores, dado que todos los vectores del grupo anterior tenían las variables de costo fijadas en cero.

Con esto, se han hallado $(n-2)^2 + m$ vectores afi en F. Por lo tanto, 5.8 define una faceta de $conv(S')$.

La desigualdad anterior define una faceta del poliedro reducido $conv(S')$, que consta de los puntos de S que tienen $y_{ik} = 1$. En lugar de considerar que estos puntos tienen esta restricción, puede pensarse que están sujetos a las condiciones equivalentes $y_{i'k} = y_{ik'} = 0$, para $i' \neq i$ y $k' \neq k, n$. Se puede realizar *lifting* sobre estas $2n-1$ variables, que si es máximo (cfr. el anexo a este capítulo) aumenta la dimensión de la cara definida en cada paso. Como se partió de una faceta, luego de los $2n-1$ pasos de *lifting*, se obtiene una faceta de $conv(S)$.

Lifting sobre $y_{i'k}$**.** Como la desigualdad tiene término libre nulo, entonces el coeficiente de *lifting* que acompaña a $y_{i'k}$ es $-\psi_{i'}$, donde

$$\psi_{i'} = \max_{y_{i'k}=1} \Big(c_{ij} - \sum_{l \in N(k)} y_{jl} - \sum_{j<i'} \psi_j y_{jk} \Big).$$

En esta expresión, suponemos que ya hicimos *lifting* sobre y_{jk} con $j < i'$. Este último término es nulo, dado que $y_{i'k} = 1$, con lo cual $y_{jk} = 0$ para $j \neq i'$. Debemos maximizar, entonces, $c_{ij} - \sum_{l \in N(k)} y_{jl}$, suponiendo que i' está asignada al procesador k. Ahora bien, esta expresión puede valer a lo sumo 1, y se alcanza este valor si se asignan i y j a procesadores conectados, de modo tal que $\sum_{l \in N(k)} y_{jl} = 0$. Esto es posible siempre que el procesador k no sea vecino de todos los restantes procesadores *no aislados* (es decir, $d(k) < n-1$ si ningún procesador está aislado). Entonces, el coeficiente que acompaña a la variable $y_{i'k}$ es -1, siempre que se cumpla la condición anterior.

Lifting sobre $y_{ik'}$**.** A partir de la desigualdad resultante en los pasos anteriores, realizamos *lifting* máximo sobre las variables $y_{ik'}$. En este caso, debemos maximizar

$$\psi_{k'} = \max_{y_{ik'}=1} \Big(c_{ij} - \sum_{l \in N(k)} y_{jl} - \sum_{j \neq i} \psi_j y_{jk} - \sum_{l<k'} \psi_l y_{il} \Big).$$

Como $y_{ik'} = 1$, entonces $y_{il} = 0$ si $l \neq k'$, y el cuarto término es nulo. Por otra parte, $\psi_j = 1$ y $\sum_{j \neq i} y_{jk} = 1$, dado que alguna tarea $\neq i$ se debe asignar al procesador k, pues i se asigna sobre k'. Entonces, la expresión a maximizar es:

$$\psi_{k'} = \max_{y_{ik'}=1} \Big(c_{ij} - \sum_{l \in N(k)} y_{jl} - 1 \Big).$$

Del mismo modo que en el caso anterior, sabemos que la tarea i se asigna sobre el procesador k', y buscamos asignar la tarea j a un vecino de k', de modo tal de lograr $c_{ij} = 1$. Sin embargo, si $N(k') \subseteq N(k)$, la sumatoria $\sum y_{jl}$ vale 1, y entonces $\psi_{k'} = -1$. Tenemos, entonces:

$$\psi_{k'} = \begin{cases} -1 & \text{si } N(k') \subseteq N(k) \\ 0 & \text{en caso contrario} \end{cases}$$

Luego de estos pasos, obtenemos la desigualdad

$$c_{ij} \leq \sum_{l \in N(k)} y_{jl} + \sum_{j \neq i} y_{jk} + \sum_{l < k'} \psi_l y_{il}.$$

Como en cada paso, el *lifting* es máximo, entonces las desigualdades intermedias definen facetas de sus respectivos poliedros. Como la desigualdad resultante es válida para todos los puntos de S, entonces define una faceta de $conv(S)$.

***

Recordemos la desigualdad 5.7, que es válida cuando $y_{ik} = 1$:

$$\sum_{j \in N(i)} c_{ij} \leq \sum_{j \in N(i)} \sum_{l \in N(k)} y_{jl}.$$

A partir de esta desigualdad válida sólo para algunos puntos, se obtuvieron dos desigualdades mediante distintas aplicaciones del procedimiento de *lifting*. Se describe a continuación una nueva desigualdad obtenida con este mismo procedimiento, pero aplicado a una pequeña variación de 5.7.

Como $\sum_{l \in W} y_{jl} + \sum_{l \notin W} y_{jl} = 1$, entonces 5.7 puede escribirse como

$$\sum_{j \in N(i)} c_{ij} \leq d(i) - \sum_{j \in N(i)} \sum_{l \notin N(k)} y_{jl}. \tag{5.9}$$

Nuevamente, esta desigualdad es válida para S', es decir, cuando $y_{ik} = 1$ (o, equivalentemente, cuando $y_{ik'} = 0$, para $k' \neq k$). Aplicamos el procedimiento de *lifting* sobre las variables $y_{ik'}$. En cada paso, debemos resolver:

$$\psi_{k'} = \max_{y_{ik'}=1} \Big(\sum_{j \in N(i)} c_{ij} + \sum_{l \notin N(k)} \sum_{j \in N(i)} y_{jl} \Big).$$

Esta expresión contiene también el término $\sum \psi_l y_{il}$, que proviene de los pasos de *lifting* anteriores, pero esta sumatoria es nula dado que $y_{ik'} = 1$. Para hallar expresiones cerradas para este máximo, llamemos $t(k') = |N(k') \backslash N(k)|$, y consideremos dos casos:

- Si $t(k') \geq d(i)$, entonces pueden asignarse todas las tareas vecinas de i a procesadores de $N(k') \backslash N(k)$, con lo cual $t = \sum c_{ij} + \sum \sum y_{jl} = 2d(i)$. Como $t \leq 2d(i)$, entonces $\psi_{k'} = 2d(i)$.

- Si $t(k') < d(i)$, no se pueden asignar todas las tareas vecinas de i a procesadores de $N(k')\backslash N(k)$, con lo cual la cota $t \leq 2d(i)$ no puede alcanzarse. Para obtener el máximo, se asigna la mayor cantidad de vecinos de i a procesadores de $N(k')\backslash N(k)$, continuando luego con los restantes vecinos de k'. No es difícil verificar que este modo de asignar las tareas permite alcanzar el valor máximo. Luego, en este caso se tiene que

$$\psi_{k'} = t(k') + \min\ \{d(i), d(k')\}.$$

La desigualdad resultante del proceso de *lifting* es

$$\sum_{j \in N(i)} c_{ij} + \sum_{l \neq k} (d(i) - \psi_l) y_{il} \leq d(i) - \sum_{l \notin N(k)} \sum_{j \in N(i)} y_{jl},$$

con ψ_l definidos según los dos casos anteriores. Esta desigualdad es válida para todas las soluciones factibles enteras del problema.

5.4 Familias exponenciales de desigualdades

Esta sección contiene las familias de desigualdades halladas que están formadas por un número exponencial de elementos. Se prueba, además, que las tres primeras familias definen facetas de $conv(S)$. En estos casos, es importante contar con procedimientos de separación adecuados, dado que el análisis exhaustivo de todas sus desigualdades es computacionalmente imposible. Se describen en esta sección las familias, mientras que los procedimientos de separación correspondientes se detallan en el próximo capítulo.

***

Cuando se resuelve la relajación lineal del modelo planteado, no se obtienen en general soluciones enteras. Más aún, en muchos casos las soluciones halladas en los ejemplos que se probaron no son cortadas por las desigualdades anteriores. Los óptimos de la relajación lineal que no son enteros tienen una cierta estructura, que puede aprovecharse para generar desigualdades válidas que los corten. Se presenta en esta sección una desigualdad válida que corta estas soluciones bajo condiciones generales.

Cuando se resuelve la relajación lineal del problema, se observa en muchos casos el siguiente hecho: Las tareas no se asignan a un solo procesador, sino que se "reparten" entre distintos procesadores (por ejemplo, la tarea i se reparte entre los procesadores 1, 2 y 3 del siguiente modo: $y_{i1} = y_{i2} = y_{i3} = 1/3$, y las restantes $y_{ij} = 0, j > 3$). El "espacio" que deja la tarea en estos procesadores es ocupado a su vez por otras tareas, y en muchos casos se forman "cadenas" de tareas que completan entre sí el espacio en los procesadores. Esto sucede con el objetivo de permitir que muchas (o todas) las variables c_{ij} tomen el valor 1, logrando así un óptimo mayor que el óptimo entero. Por lo tanto, un corte para estos valores debería acotar superiormente a los valores c_{ij}, aislados o en combinación. La desigualdad que se presenta en este parágrafo realiza esta tarea en muchos casos.

Proposición. *Si $i \in V(G)$, y $k, k' \in V(H)$, entonces*

$$c_{ij} \leq \sum_{l \in N(k) \cup N(k')} y_{jl}$$

es desigualdad válida para todos los puntos de S en los cuales $y_{ik} + y_{ik'} = 1$.

Demostración. Sea $(y, c) \in S'$, y supongamos, por ejemplo, que está asignada al procesador k (el otro caso es similar). Si la tarea j se asigna a algún vecino de k, entonces el lado derecho de la desigualdad es 1, y la desigualdad se escribe como $c_{ij} \leq 1$, que siempre se cumple. Por otra parte, si la tarea j no se asigna a ningún vecino de k, entonces $c_{ij} = 0$, con lo cual la desigualdad toma la forma $0 \leq \sum y_{jl}$, que se verifica en todos los casos.

Con esto, se tiene una desigualdad válida para $S \cap \{(y, c) : y_{ik} + y_{ik'} = 1\}$. Esta desigualdad no es trivial cuando $y_{ik} + y_{ik'} = 1$, lo cual sucede cuando la tarea i se asigna a alguno de estos procesadores, pero también sucede cuando se halla "repartida" entre los dos procesadores (por ejemplo, con $y_{ik} = y_{ik'} = 1/2$). Es en este último caso cuando constituye un corte.

Supongamos, entonces, que tenemos un punto (y, c) en el cual $y_{ik} = y_{ik'} = 1/2$. Las tareas $j \in N(i)$ que se encuentren asignadas a procesadores en $N(k) \cup N(k')$ tendrán $\bar{c}_{ij} = 1$ en el óptimo, pero sucede con frecuencia que estas tareas a su vez están repartidas entre más de un procesador, con lo cual "sólo una parte" de ellas está en los procesadores de $N(k) \cup N(k')$, y entonces $\sum_{l \in N(k) \cup N(k')} y_{jl} < 1$. Si esto sucede con al menos una tarea $j \in N(i)$, entonces la desigualdad correspondiente corta a este punto.

Esta desigualdad considera a la tarea i cuando está repartida entre dos procesadores (los procesadores k y k'). Sin embargo, puede extenderse para considerar conjuntos de procesadores. Sea $W \subseteq V(H)$, con $|W| \geq 2$, y sea $N(W) := \{l \in V(H) : kl \in E(H)$, para algún $k \in W\}$. La siguiente proposición se prueba del mismo modo que la anterior.

Proposición. *Si $i \in V(G)$, y $W \subseteq V(H)$, entonces*

$$c_{ij} \leq \sum_{l \in N(W)} y_{jl} \tag{5.10}$$

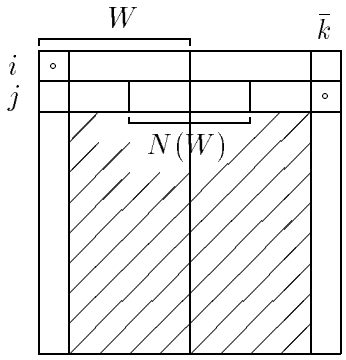
es válida para todos los puntos de $S' = S \cap \{(y, c) : \sum_{k \in W} y_{ik} = 1\}$.

Debe notarse que también puede probarse esta proposición por argumentos disyuntivos, considerando las distintas desigualdades generadas al asignarse la tarea i a cada procesador de W, y tomando el mínimo entre los coeficientes de cada variable. Con esto, también se prueba que esta desigualdad es una D-desigualdad[5].

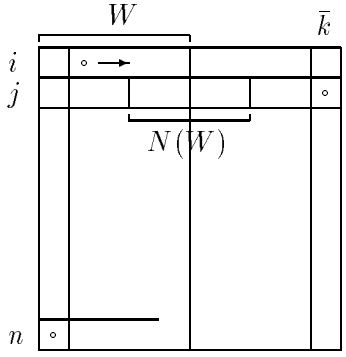
Teorema. *Sea F la cara de $conv(S')$ definida por la desigualdad 5.10. Entonces, F tiene dimensión al menos $n^2 - 3n + |W| + m$.*

[5] Si $\pi^1 x \leq \pi_0^1$ es válida para S^1 y $\pi^2 x \leq \pi_0^2$ es válida para S^2, con $S = S^1 \cup S^2$, entonces $\sum_i \min\{\pi_i^1, \pi_i^2\} x_i \leq \max\{\pi_0^1, \pi_0^2$ es una *D-desigualdad*, válida para S.

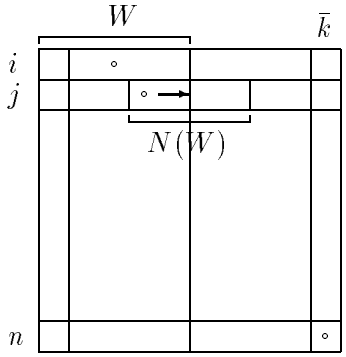
Demostración. Se deben hallar $n^2 - 3n + |W| + m + 1$ vectores afi en S' que cumplan 5.10 por igualdad. Supongamos que existe algún procesador $\bar{k} \in N(W) \backslash W$ (la demostración es similar si W tiene por vecinos a todos los restantes procesadores), y supongamos también que $W = \{1, 2, ..., |W|\}$.



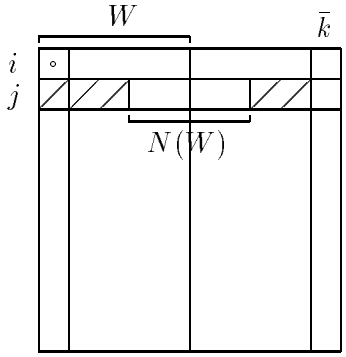
El primer grupo de puntos se obtiene asignando la tarea i al primer procesador de W y la tarea j sobre $\bar{k}$. Las restantes variables (líneas diagonales) forman un problema de asignación de $n-2$ tareas, con lo cual esta parte de la solución puede completarse con $(n-3)^2 + 1$ vectores afi. En cada uno, se ajusta la variable c_{ij} en cero para cumplir 5.10 por igualdad (i y j se asignan en procesadores no conectados). Las otras variables se fijan en cero.



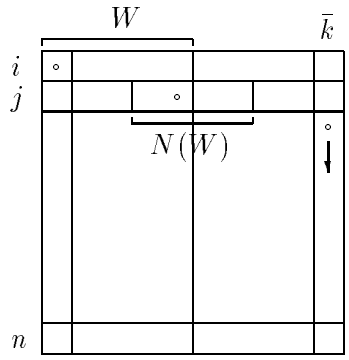
El segundo grupo de vectores está formado por $|W| - 1$ puntos, obtenidos asignando la tarea j sobre $\bar{k}$ y la tarea i sobre cada procesador $k \in W$, $k \neq 1$. Se fija c_{ij} en cero, para cumplir 5.10 por igualdad. Además, estos vectores son afi entre si y con los anteriores, en los cuales $y_{ik} = 0$.



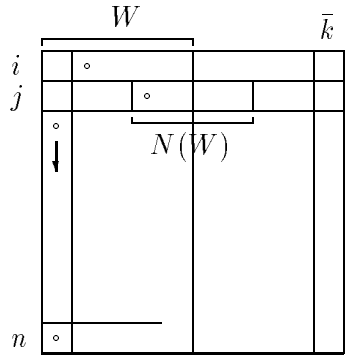
Se forman ahora $|N(W)|$ nuevos vectores, moviendo la tarea j sobre todos los procesadores $l \in N(W)$. Además, se asigna la tarea i sobre algún procesador $k' \in W$ vecino de l. De esta forma, se puede fijar $c_{ij} + 1$, cumpliendo 5.10 por igualdad. Cada nuevo punto de este grupo es afi con los anteriores (de este grupo y de los precedentes), dado que en todos ellos se tiene $y_{il} = 0$.



El cuarto grupo de vectores se obtiene fijando la tarea i sobre el primer procesador de W, y asignando la tarea j sobre todos los procesadores de $N(W)$. De esta forma, se tienen $n - |N(W)| - 1$ nuevos vectores, en los cuales $c_{ij} = 0$, cumpliendo 5.10 por igualdad, y que son afi con los anteriores. Debe notarse que si $1 \in N(W)$, entonces cuando se asigne la tarea j sobre este procesador debe correrse la tarea i a otro procesador de W, distinto del primero.



Fijemos ahora la tarea i sobre el primer_procesador de W, y asignemos sobre el procesador k todas las tareas $t \in V(G) \backslash \{i, j, n\}$. La tarea j se asigna sobre algún procesador vecino de 1, de modo tal de fijar $c_{ij} = 1$, cumpliendo 5.10 por igualdad. Cada uno de estos vectores tiene $y_{t\bar{k}} = 1$, con lo cual es afi con los anteriores, en los que esta variable tiene valor nulo.



Por último, consideremos los $n-3$ vectores que se obtienen asignando sobre el primer procesador de W todas las tareas $t \in \backslash \{i, j, n\}$. La tarea i se asigna sobre el procesador $2 \in W$ y la tarea j se fija sobre algún vecino de este procesador, de modo tal de permitir que $c_{ij} = 1$, cumpliendo 5.10 por igualdad.

En todos los puntos anteriores, las variables de costo, excepto c_{ij}, son nulas. Para completar la dimensión de la cara, consideremos $m-1$ nuevos puntos, uno por cada eje $rs \neq ij$. En cada uno de estos puntos, se asigna la tarea i sobre algún procesador de W, y la tarea j sobre un vecino de este procesador, de modo tal de poder fijar $c_{ij} = 1$ para cumplir 5.10 por igualdad. Las restantes tareas se asignan en forma arbitraria, cuidando que las tareas r y s se ubiquen en procesadores conectados Todas las variables de costo se fijan en cero, excepto c_{rs}, que se fija en 1. De esta forma, se tienen $m-1$ nuevos vectores afi con los anteriores.

Por lo tanto, se tiene la siguiente cantidad de vectores:

Grupo 1:	n^2	$-6n$				$+10$
Grupo 2:					$\|W\|$	-1
Grupo 3:				$\|N(W)\|$		
Grupo 4:		n		-$\|N(W)\|$		-1
Grupo 5:		n				-3
Grupo 6:		n				-3
Grupo 7:			m			-1
	n^2	$-3n$	$+m$		$+\|W\|$	$+1$

Se tienen, entonces, $n^2-3n+m+|W|+1$ vectores afi que cumplen 5.10 por igualdad. Por lo tanto, esta desigualdad define una cara de $conv(S')$ de dimensión al menos $n^2-3n+m+|W|$.

La desigualdad 5.10 define una faceta de $conv(S')$ para aquellos conjuntos W de procesadores para los cuales $N'(k_i) \neq N(k_i)$. Aunque esta condición puede parecer restrictiva, debe notarse que es cumplida por un número exponencial de subconjuntos de procesadores

(en las arquitecturas habituales), con lo cual se ha identificado una familia exponencial de facetas de $conv(S')$. Por otra parte, esta desigualdad es válida sólo para los puntos de S que tienen $\sum_{k \in W} y_{ik} = 1$. Puede pensarse en aplicar el proceso de *lifting* para obtener una desigualdad válida para S, mejorando al mismo tiempo la dimensión de la cara definida. Se describe a continuación la aplicación de este procedimiento a la presente desigualdad.

Se realiza el proceso de *lifting* sobre las variables $y_{ik} (k \notin W)$ en orden creciente de la variable k. Cuando realizamos *lifting* de la variable $y_{ik} (k \notin W)$, ya lo hemos hecho con las variables $y_{il} (l \notin W, l < k)$, y hemos obtenido los coeficientes α_l respectivos. En el paso correspondiente a la variable y_{ik}, debemos encontrar el siguiente máximo:

$$\psi_k = max_{y_{ik}=1} \Big(c_{ij} - \sum_{l \in N(W)} y_{jl} + \sum_{l \notin W, l<k} \alpha_l y_{il} \Big).$$

Analizando este máximo, puede verse que el tercer término es nulo, dado que $y_{ik} = 1$ (con lo cual $y_{il} = 0$), y entonces el *lifting* será independiente de la secuencia elegida para las variables. El primer término puede maximizarse asignando la tarea j en un procesador de $N(k)$. Sin embargo, si esta tarea se asigna en $N(k) \cap N(W)$ también restará una unidad al aparecer en el segundo término. Por estos motivos, el máximo tiene el valor

$$\psi_k = \begin{cases} 1 & \text{si } N(k) \backslash N(W) \neq \emptyset \\ 0 & \text{si } N(k) \backslash N(W) = \emptyset \end{cases}$$

Como el término independiente de la desigualdad es cero, entonces el coeficiente que acompaña a la variable y_{ik} es $\alpha_k = -\psi_k$. Por lo tanto, la desigualdad resultante es la siguiente:

$$c_{ij} \leq \sum_{l \in W} y_{jl} + \sum_{k \notin W} \psi_k y_{ik} \tag{5.11}$$

En S' las variables y_{ik} $(k \notin W)$ están forzadas a ser cero. El procedimiento de *lifting* se realizó sobre estas variables, asegurando que la nueva desigualdad es válida para todo el conjunto S. Por otra parte, como el *lifting* ha sido máximo, entonces la dimensión de la cara definida aumenta en cada paso. Se han realizado $n - |W|$ pasos, con lo cual la dimensión de la cara definida por 5.11 es de $n^2 - 2n + m = (n-1)^2 + m - 1$. Con esto, se prueba el siguiente teorema.

Teorema. *La desigualdad 5.11 define una faceta de* $conv(S)$.

Como esta desigualdad está definida por un eje de G y un subconjunto de procesadores, entonces se tiene una familia de facetas formada por una cantidad exponencial de desigualdades. Por otra parte, se vio que estas desigualdades constituyen cortes bajo condiciones generales, con lo cual son de gran utilidad para los algoritmos basados en planos de corte.

Pueden generalizarse estas desigualdades, para considerar conjuntos de ejes en lugar de hacerlo con ejes individuales. Si en lugar de acotar la variable c_{ij}, como se hizo para esta

desigualdad, se considera un subconjunto $T \subseteq N(i)$ de vecinas de i, entonces la siguiente desigualdad es válida para S':

$$\sum_{j \in T} c_{ij} \leq \sum_{j \in T} \sum_{k \in N(W)} y_{jk} \tag{5.12}$$

Del mismo modo que en el caso anterior, puede realizarse *lifting* máximo sobre las variables y_{ik} $(k \notin W)$, que están fijas en cero. Sea $t(k) = |N(k) \backslash N(W)|$. Los coeficientes de *lifting* son ahora los siguientes:

$$\psi_k = \begin{cases} |T| & \text{si } t(k) \geq |T| \\ t(k) & \text{si } t(k) < |T| \end{cases}$$

Sin embargo, no fue posible probar que estas nuevas desigualdades definen facetas. Por el contrario, la dimensión de la cara definida por estas desigualdades parece disminuir a medida que se consideran conjuntos T de mayor tamaño, como lo muestra la siguiente proposición.

Proposición. *Sea F la cara definida por 5.12 luego del proceso de lifting. Entonces $dim(F) \geq dim(S) - |T|$.*

Demostración. La demostración procede del mismo modo que se hizo con 5.10, excepto en el último grupo de vectores, que sólo se puede formar con $m - |T| - 1$ puntos afi entre sí y con los anteriores. De esta forma, luego del proceso de *lifting* se obtienen $(n-1)^2 + m - |T| + 1$ vectores afi, con lo cual la cara tiene dimensión al menos $(n-1)^2 + m - |T|$.

***

Las restricciones de costo del modelo

$$c_{ij} + y_{ik} \leq 1 + \sum_{l \in N(k)} y_{jl}.$$

definen el valor de las variables c_{ij} en función de las variables y_{ik}, y se prueba que definen facetas bajo condiciones generales. Estas restricciones consideran procesadores k individuales, y acotan el valor de cada variable c_{ij}. En la familia de desigualdades que se presenta a continuación, se reemplaza este procesador k por un conjunto no vacío de procesadores.

Proposición. *Sea $ij \in E(G)$ un eje del grafo de tareas y sea $W \subseteq V(H)$ un subcojunto no vacío de procesadores. Entonces, la siguiente desigualdad es válida:*

$$c_{ij} + \sum_{k \in W} y_{ik} \leq 1 + \sum_{l \in N(W)} y_{jl} \tag{5.13}$$

Demostración. Sea $(y, c) \in S$ una solución factible entera. Si $\sum_{k \in W} y_{ik} = 0$ (es decir, si la tarea i se asigna fuera del conjunto W), entonces la desigualdad se cumple trivialmente,

dado que toma la forma $c_{ij} \leq 1 + \sum y_{jl}$. Por otra parte, si $\sum_{k \in W} y_{ik} = 1$, entonces la tarea i se asigna sobre algún procesador k dentro del conjunto, y la desigualdad toma la forma $c_{ij} \leq \sum y_{jl}$. En este caso, la desigualdad puede no cumplirse sólo cuando $\sum y_{jl} = 0$, pero si esto sucede, entonces la tarea j se asigna sobre algún procesador no vecino de k, con lo cual $c_{ij} = 0$. Luego, 5.13 es desigualdad válida para S.

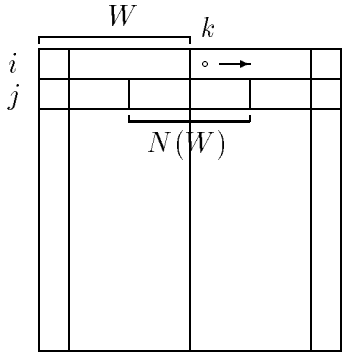
Esta desigualdad está definida por un eje ij y por un subconjunto W de procesadores, con lo cual se tiene un número exponencial de desigualdades en esta familia. Del mismo modo que se probó que las restricciones de costo del modelo definen facetas, puede verse que estas desigualdades también lo hacen, modificando levemente la condición bajo la cual esto sucede. Las restricciones de costo definen facetas si $N(l) \not\subseteq N(k)$ para todo procesador $l \neq k$, mientras que sucede esto mismo con las nuevas desigualdades 5.13 cuando $N(l) \not\subseteq N(W)$ [6].

Proposición. *Supongamos que $N(l) \not\subseteq N(W)$ para todo procesador $l \notin W$. Entonces, 5.13 define una faceta de $conv(S)$.*

Puede verificarse que bajo estas hipótesis, la desigualdad 5.13 es la misma que resulta del proceso de *lifting* sobre 5.10, descripto en el parágrafo anterior. Se muestra ahora una forma de reforzar estas desigualdades para que definan facetas en todos los casos. Sean $P := \{k \notin W : N(k) \backslash N(W) \text{ es vacío } \}$, y $S_P := S \cap \{(y, c) : y_{ik} = 0 \text{ para } k \in P\}$.

Teorema. *La desigualdad 5.13 define una cara de $conv(S_P)$ de dimensión al menos $(n-1)^2 + m - |P|$.*

Demostración. La demostración procede del mismo modo que con el primer teorema del parágrafo anterior. Los puntos afi que se construyeron en la demostración de este teorema también cumplen 5.13 por igualdad, dado que en todos ellos $\sum_{k \in W} y_{ik} = 1$, con lo cual 5.13 se escribe del mismo modo que 5.10. Entonces, esta desigualdad define una cara de dimensión al menos $n^2 - 3n + m + |W|$.



En todos estos puntos, se tiene que $y_{ik} = 0$ para $k \notin W$. Construimos ahora $n - |W| - |P|$ nuevos puntos afi con los anteriores y que cumplen 5.13 por igualdad. Consideremos cada procesador $k \notin P$, y asignemos la tarea i sobre k y la tarea j sobre $l \in N(k) \backslash N(W)$ (que es no vacío pues $n \notin P$). Con esto, puede fijarse $c_{ij} = 1$. Estos vectores son afi entre ellos y con los anteriores, en los cuales $y_{ik} = 0$.

Se tiene entonces que 5.13 define una cara de $conv(S_P)$ de dimensión por lo menos $(n-1)^2 + m - |P|$.

[6] Debe notarse que esta condición es cumplida por un número exponencial de conjuntos de procesadores, si se consideran las arquitecturas habituales.

Podemos aplicar ahora el procedimiento de *lifting* sobre las variables y_{ik} $(k \in P)$, que están fijas en cero en S_P. El coeficiente que acompaña a la variable y_{ik} es $\alpha_k = 1 - \psi_k$, donde

$$\psi_k := \max_{y_{ik}=1}\Big(c_{ij} + \sum_{l \in W} y_{il} - \sum_{l \in N(W)} y_{jl} + \sum_{l \in P, l<k} \alpha_l y_{il}\Big).$$

En esta expresión, el término $\sum_{k \in W} y_{ik}$ es nulo, dado que $y_{ik} = 1$ y $k \notin W$. Además, $\sum_{l<k} y_{il} = 0$, por las mismas razones. Entonces, la expresión a maximizar es la siguiente:

$$\psi_k := \max_{y_{ik}=1}\Big(c_{ij} - \sum_{l \in N(W)} y_{jl}\Big).$$

Ahora bien, como $k \in P$, entonces $N(k) \subseteq N(W)$, con lo cual siempre que la tarea j se asigne sobre un vecino de k, se tendrá que $\sum_{l \in N(W)} = 1$. Por lo tanto, $\psi_k \leq 0$, y este valor se obtiene asignando j sobre algún vecino de k. Por lo tanto, $\psi_k = 0$ y $\alpha_k = 1$. Con esto, hemos probado el siguiente teorema:

Teorema. *La siguiente desigualdad define una faceta de* $conv(S)$*:*

$$c_{ij} + \sum_{k \in W \cup P} y_{ik} \leq 1 + \sum_{l \in N(W)} y_{jl}.$$

Del mismo modo que en el parágrafo anterior, se ha identificado una segunda familia exponencial de facetas para el poliedro de *mapping*.

***

Consideremos dos subconjuntos disjuntos W y W' de procesadores ($|W| \geq 2$ y $|W'| \geq 2$), de modo tal que no exista ningún eje entre W y W'. Sea $i \in V(G)$ una tarea y sean $j, j' \in N(i)$ dos tareas vecinas. Supongamos que la tarea i se asigna sobre W y j' se asigna sobre W'. En este caso, la siguiente desigualdad es válida:

$$c_{ij} + c_{ij'} \leq \sum_{k \in N(W)} y_{jk} \tag{5.14}$$

Como no existe ningún eje entre W y W', y las tareas i y j' se asignan sobre W y W' respectivamente, entonces $c_{ij'} = 0$, y c_{ij} puede valer 1 sólo cuando j se asigna sobre un vecino de W, con lo cual esta desigualdad es válida para $\bar{S} := \{(y, c) \in S : \sum_{k \in W} y_{ik} = 1, \sum_{k \in W'} y_{j'k} = 1\}$.

Del mismo modo que con la familia anterior, se puede realizar el proceso de *lifting* en tiempo polinomial para obtener una nueva desigualdad válida para S. Se describe primero este proceso, para pasar luego al análisis de la dimensión de la cara que esta desigualdad define. Puede pensarse al conjunto $\bar{S}$ como el conjunto S, con las restricciones adicionales $y_{ik} = 0$ para $k \notin W$, e $y_{j'k} = 0$ para $k \notin W'$. El proceso de *lifting* se realiza sobre estas variables.

Lifting sobre y_{ik} ($k \notin WS$). Como la desigualdad 5.14 tiene término independiente igual a cero, entonces el coeficiente de *lifting* (en caso de ser máximo) será $-\psi_k$, donde

$$\psi_k := \max_{y_{ik}=1} \Big(c_{ij} + cij' - \sum_{k \in N(W)} y_{jk} \Big).$$

En la expresiíon a maximizar sólo participan las tareas j y j' (la tarea i está asignada al procesador k), con lo cual debe hallarse la mejor forma de asignar estas dos tareas. El término $c_{ij} - \sum y_{jk}$ aporta 0 si $N(k) \subseteq N(W)$ y 1 en caso contrario, y el término $c_{ij'}$ puede valer 1 si existe en W' algún vecino de k (en cuyo caso se asigna j' a este procesador). Por lo tanto, se tiene que:

$$\psi_k = \begin{cases} 0 & \text{si } N(k) \subseteq N(W) \text{ y } N(k) \bigcap W' = \emptyset \\ 2 & \text{si } N(k) \not\subseteq N(W) \text{ y } N(k) \bigcap W' \neq \emptyset \\ 1 & \text{en caso contrario} \end{cases}$$

Lifting sobre $y_{j'k}$ ($k \notin W'$). El coeficiente de *lifting* que acompaña a la variable $y_{j'k}$ es $-\psi'_k$, donde

$$\psi'_k := \max_{y_{j'k}=1} \Big(c_{ij} + cij' - \sum_{k \in N(W)} y_{jk} - \sum_{k \notin W} \psi_k y_{ik} \Big).$$

Este máximo se calcula sobre todos los puntos en los cuales la tarea j' se asigna al procesador k, y sólo depende de la ubicación de las tareas i y j. Entonces, puede obtenerse ψ'_k por inspección de estos $O(n^2)$ puntos. No es sencillo derivar una expresión para ψ'_k dada la cantidad de casos posibles, pero el procedimiento continúa siendo polinomial.

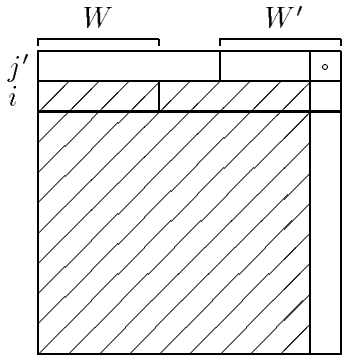
Luego del proceso de *lifting*, se obtiene la siguiente desigualdad, que es válida para S:

$$c_{ij} + cij' \leq \sum_{k \in N(W)} y_{jk} + \sum_{k \notin W} \psi_k y_{ik} + \sum_{k \notin W'} \psi'_k yj'k \tag{5.15}$$

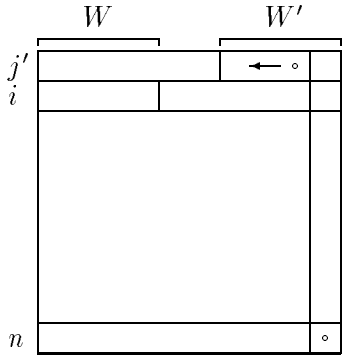
El análisis de la dimensión de la cara definida por esta desigualdad es muy similar al realizado para la desigualdad anterior, salvo una pequeña diferencia en el proceso de *lifting*. La siguiente proposición se prueba de modo similar a las anteriores.

Proposición. *Sea F la cara de $\bar{S}$ definida por 5.14. Entonces, $dim(F) \geq n^2 - 4n + m + |W| + |W'| - 1$.*

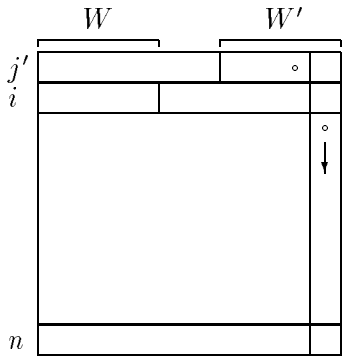
Demostración. En los siguientes diagramas, suponemos que los procesadores de W son los $|W|$ primeros (ubicados a la izquierda) y que los procesadores de W' son los $|W'|$ últimos (sobre la derecha).



Si asignamos la tarea j' sobre el último procesador de W', tenemos un problema con $n-1$ tareas (marcado por las líneas oblicuas). Además, se fija $c_{ij'}=0$, con lo cual 5.14 es igual a 5.10, y por el teorema de la dimensión de 5.10, se tienen $(n-1)^2-3(n-1)+|W|+1$ vectores afi que cumplen 5.10 por igualdad (no consideramos los $m-1$ vectores del último grupo). Si extendemos estos vectores al espacio de n tareas y n procesadores del problema actual (fijando j' sobre el último procesador, se tiene la misma cantidad de vectores afi que cumple 5.14 por igualdad.



Fijemos ahora la tarea $n \neq i, j, j'$ en el último procesador de W' y asignemos la tarea j' sobre todos los restantes procesadores de este conjunto. Las tareas i y j se asignan arbitrariamente (i dentro de W) y se ajusta c_{ij} para cumplir 5.14 por igualdad. De esta forma, se tienen $|W'|-1$ nuevos vectores afi entre si y con los anteriores.



Consideremos los $n-3$ vectores que se forman al asignar sobre el último procesador de W' las tareas de $V(G)\backslash\{i, j', n\}$. Nuevamente, estos vectores son afi entre si y con los anteriores.

En los vectores de estos tres grupos, las variables de costo se fijan en cero, excepto c_{ij}. Podemos ahora considerar $m-2$ nuevos vectores, uno para cada eje rs, con $rs \neq ij$ y $rs \neq ij'$, asignando r y s sobre procesadores conectados (y manteniendo i en W y j' en W'), de modo tal que $c_{rs}=1$ y las restantes variables de costo se fijan en cero. Cada nuevo punto de este grupo es afi con los anteriores, dado que en todos ellos esta variable de costo es nula. Se han hallado, entonces, $n^2-4n+m+|W|+|W'|$ vectores, lo cual completa la demostración.

Se realizó el proceso de *lifting* sobre $2n-|W|-|W'|$ variables, con lo cual se tendría que la dimensión de la cara definida por 5.15 sería de $(n-1)^2+(m-2)$ (es decir, una dimensión menos que la necesaria para definir una faceta). Sin embargo, es posible realizar otro análisis, que prueba que 5.15 define efectivamente una faceta de $conv(S)$.

Teorema. *Supongamos que existe un procesador $l \notin W$ tal que $k_1 l \in E(H)$ para algún $k_1 \in W'$ y $k_2 l \notin E(H)$ para algún $k_2 \in W'$. Supongamos además que el primer paso de lifting se realiza sobre la variable y_{il}. Entonces, 5.15 define una faceta de $conv(S)$.*

Demostración. De acuerdo con la proposición anterior, se tienen $x^0, x^1, ..., x^t$ vectores afi en S' que cumplen 5.14 por igualdad ($t = n^2 - 4n + m + |W| + |W'|$). Como estos puntos se encuentran en S', entonces cumplen $y_{il} = 0$, y como no existe ningún eje entre W y W', entonces también cumplen $c_{ij'} = 0$. Luego del primer paso de *lifting*, se tiene una desigualdad que es válida también para los puntos en los cuales $y_{il} = 1$. Construimos ahora dos nuevos puntos x^{t+1} y x^{t+2} que son afi entre si y con los anteriores.

1. El punto x^{t+1} se obtiene asignando la tarea i al procesador l y la tarea j' al procesador k_2, de modo tal que $c_{ij'} = 0$. Las restantes tareas se asignan en forma arbitraria, ajustando c_{ij} según j se asigne a un vecino de l o no. Este vector es afi con los anteriores, dado que en x^i $(0 \leq i \leq t)$ la variable y_{il} toma el valor cero.
2. El punto x^{t+2} se obtiene asignando la tarea i al procesador l y la tarea j' al procesador k_1 (las restantes variables se fijan del mismo modo que en el caso anterior). Ahora, se tiene un punto en el cual $c_{ij'} = 1$ por primera vez, con lo cual es afi con los anteriores.

Se tienen, entonces, $t + 3$ vectores afi luego del primer paso de *lifting*. Como el *lifting* es máximo en todos los pasos siguientes, se tiene que todo el proceso aumenta en $2n - |W| - |W'| + 1$ la dimensión de la cara definida por 5.14. Por lo tanto, existen $(n^2 - 4n + m + |W| + |W'|) + (2n - |W| - |W'| + 1) = (n^2 - 2n + 1) + m = (n - 1)^2 + m$ puntos afi que cumplen 5.15 por igualdad, con lo cual la desigualdad 5.15 define una cara de dimensión $(n - 1)^2 + m - 1$. Por lo tanto, esta desigualdad define una faceta.

***

En el modelo planteado se tienen las restricciones $c_{ij} \leq 2-(y_{ik}+y_{jl})$, donde $ij \in E(G), kl \notin E(H)$. Estas restricciones indican que $c_{ij} = 0$ si las tareas i y j se asignan a procesadores no concetados, y $c_{ij} \leq 1$ en caso contrario. En esta sección, se presenta una generalización de estas restricciones, considerando conjuntos independientes en lugar de pares de nodos no conectados. Se verá luego que las desigualdades obtenidas pertenecen a un grupo más amplio de desigualdades válidas.

Si $T \subseteq V(G)$ es un conjunto de nodos de un grafo, llamamos $E(T)$ al conjunto de ejes de $E(G)$ entre nodos de T. Es decir, $E(T) = \{ij \in E(G) : i, j \in T\}$. Un subconjunto de nodos de un grafo es *independiente* si ningún par de nodos del conjunto está unido por un eje del grafo.

Proposición. *Sea $W \subseteq V(H)$ un conjunto independiente, y sea $T \subseteq V(G)$ un conjunto de tareas. Entonces*

$$\sum_{ij \in E(T)} c_{ij} \leq |E(T)|(|T| - \sum_{k \in W} \sum_{i \in T} y_{ik}) \tag{5.16}$$

es desigualdad válida para S.

Demostración. Cuando todas las tareas en T están asignadas a procesadores en W, entonces $|T| - \sum \sum y_{ik} = 0$, con lo cual la desigualdad se puede escribir como $\sum_{ij \in E(T)} c_{ij} \leq 0$. Las

soluciones en las cuales todas las tareas de T están en procesadores de W cumplen esta condición, dado que W es un conjunto independiente, y ningún par de tareas de T puede comunicarse. Por otra parte, si alguna tarea de T no está asignada a un procesador de W, entonces $|T| - \sum\sum y_{ik} \geq 1$, y la desigualdad se cumple siempre, dado que $\sum_{ij \in E(T)} c_{ij} \leq |E(T)|$.

En el caso particular $T = \{i, j\}$ y $W = \{k, l\} (kl \notin E(H))$, esta desigualdad domina a las restricciones del modelo inicial, dado que $E(T) = \{ij\}$. En este caso, la desigualdad se escribe como:

$$c_{ij} \leq 2 - (y_{ik} + y_{il} + y_{jk} + y_{jl}),$$

y esta desigualdad domina a la restricción $c_{ij} \leq 2 - y_{ik} - y_{jl}$. En la sección 5.2.1 se analizó la dimensión de la cara que define esta desigualdad.

La desigualdad presentada en la proposición anterior generaliza a algunas restricciones del modelo inicial. A su vez, esta desigualdad forma parte de un conjunto más amplio de desigualdades válidas, que se presentan en la siguiente proposición.

Proposición. *Sean $W \subseteq V(H)$ y $T \subseteq V(G)$ con $|T| = |W|$ y sea $M(T, W)$ el valor óptimo del problema de mapping con $G(T)$ y $G(W)$ (los subgrafos inducidos por W y T). Entonces,*

$$\sum_{ij \in E(T)} c_{ij} \leq M(T, W) + |E(T)|(|T| - \sum_{i \in T} \sum_{k \in W} y_{ik})$$

es desigualdad válida para S.

Demostración. Cuando todas las tareas de T están asignadas a procesadores de W, entonces la desigualdad tiene la forma $\sum c_{ij} \leq M(T, W)$, indicando que la cantidad de ejes asignados a procesadores adyacentes no debe superar el óptimo del problema. Cuando alguna tarea de T no está conectada a los procesadores de W, la desigualdad se cumple siempre.

La obtención de estas desigualdades implica conocer el resultado del problema de *mapping* para subgrafos del problema inicial. En algunos casos esto es posible, dada la estructura de los subgrafos elegidos. Por otra parte, la desigualdad sigue siendo válida si en lugar de $M(T, W)$ se ubica un cota superior de este valor.

La desigualdad 5.16 es un caso particular de esta desigualdad, cuando W es un conjunto independiente. En este caso se tiene que $M(T, W) = 0$. Otro caso en el cual puede obtenerse el valor exacto de $M(T, W)$ es el siguiente:

Corolario. *Sea $T \subseteq V(G)$ un circuito (posiblemente con cuerdas), y sea $W \subseteq V(H)$ un circuito sin cuerdas con la misma cantidad de nodos que T. Entonces $M(T, W) = |T|$, y la siguiente desigualdad es válida para S:*

$$\sum_{ij \in E(T)} c_{ij} \leq |T| + |E(T)|(|T| - \sum_{k \in W} \sum_{i \in T} y_{ik}).$$

***

Se presentó la familia de desigualdades

$$\sum_{ij \in E(T)} c_{ij} \leq |E(T)|(|T| - \sum_{k \in W} \sum_{i \in T} y_{ik}).$$

En el caso particular $T = \{i, j\}$, con $ij \in E(G)$, la desigualdad anterior se escribe como

$$c_{ij} + \sum_{l \in I} (y_{il} + y_{jl}) \leq 2,$$

donde I es un conjunto independiente. Se presenta a continuación una nueva desigualdad válida obtenida mediante el procedimiento de Chvatal-Gomory, aplicado sobre una variación de esta desigualdad. Dado que la desigualdad anterior es válida, se puede sumarle la desigualdad $c_{ij} \leq 1$, obteniendo la siguiente desigualdad válida:

$$2c_{ij} + \sum_{l \in I} (y_{il} + y_{jl}) \leq 3 \tag{5.17}$$

Proposición. *Sean $I_1, I_2, ..., I_k$ (k impar) conjuntos independientes, tales que $\#\{j : d \in I_j\}$ es par $(\forall d \in V(G))$. Entonces, la desigualdad*

$$kc_{ij} + 1/2 \sum_{t=1}^{k} \sum_{l \in I_t} (y_{il} + y_{jl}) \leq 3r + 1,$$

es desigualdad C-G válida para S, donde $k = 2r + 1$.

Demostración. Multiplicando por $1/2$ las desigualdades 5.17 tenemos

$$c_{ij} + 1/2 \sum_{l \in I_t} (y_{il} + y_{jl}) \leq 3/2$$

Sumando ahora sobre t, tenemos la desigualdad

$$kc_{ij} + 1/2 \sum_{t=1}^{k} \sum_{l \in I_t} (y_{il} + y_{jl}) \leq 3k/2$$

En el término $1/2 \sum \sum (y_{il} + y_{jl})$, cada variable aparece un número par de veces, por la hipótesis de la proposición. Entonces, este término toma valores enteros, con lo cual la porción izquierda de la desigualdad es entera. Por lo tanto, se puede redondear el término derecho, y como $k = 2r + 1$, entonces $\lfloor 3k/2 \rfloor = \lfloor (6r + 3)/2 \rfloor = 3r + 1$, con lo cual se obtiene la desigualdad.

Las hipótesis de la proposición solicitan que cada nodo de $V(G)$ se encuentre en un número par de conjuntos independientes, y esto sucede, por ejemplo, cuando se consieran conjuntos independientes de dos procesadores no conectados formando un circuito de $\bar{E}(H) = V(H) \times V(H) - E(H)$.

Corolario. *Sea $k_1, k_2, ..., k_{2r+1} = k_1$ un circuito de procesadores en $\bar{E}(H)$ (es decir, $k_i k_{i+1} \notin E(H)$). Entonces*

$$(2r+1)c_{ij} + \sum_{t=1}^{2r+1} (y_{it} + y_{jt}) \leq 3r + 1$$

es desigualdad válida para S.

Tomando $I_t = \{k_t k_{t+1}\} (1 \leq t \leq 2r)$ como los conjuntos independientes, se tiene que cada variable y_{ik} aparece dos veces entre los conjuntos $I_1, I_2, ..., I_{2r}$, con lo cual se anulan los coeficientes $1/2$ que se encuentran en la desigualdad general, y cada variable queda acompañada por el coeficiente 1. De este modo, se aplica la proposición anterior a este caso.

Puede aplicarse la proposición a conjuntos de circuitos de $\bar{E}(H)$, en cuyo caso el coeficiente de cada variable indica la cantidad de circuitos que pasan por el procesador.

5.5 Desigualdades adicionales

Existe un punto x^{inf} cuya función objetivo es mayor que la de cualquier *mapping*, en general. Este punto se obtiene "repartiendo" todas las tareas entre todos los procesadores ($y_{ik} = 1/n$), y asignando $c_{ij} = 1$, $\forall ij \in E(G)$. Este "punto infausto" está dentro de la relajación lineal del modelo, y $cx^{inf} = m$. Como la función objetivo puede valer a lo sumo m (dado que se tiene esta cantidad de variables c_{ij} de costo), entonces *el óptimo de la relajación lineal será siempre m*. Es importante contar con desigualdades que corten a este punto, y en esta sección se presenta una familia reducida pero que lo hace efectivamente.

Proposición. *Sea $i \in V(G)$. Entonces,*

$$\sum_{j \in N(i)} c_{ij} \leq \sum_{k \in V(H)} d(k) y_{ik}$$

es desigualdad válida para S.

Demostración. En cualquier solución factible, la tarea i está asignada a un procesador $k \in V(H)$. La expresión $\sum c_{ij}$ indica la cantidad de tareas vecinas que están asignadas a procesadores vecinos, y esta cantidad no puede ser mayor que $d(k)$ (que es la cantidad de procesadores vecinos al nodo k). Entonces,

$$\sum_{j \in N(i)} c_{ij} \leq d(k) = d(k) y_{ik} \leq \sum_{k} d(k) y_{ik}.$$

Esta desigualdad corta al punto mencionado anteriormente si G no es regular[7], tomando como tarea i uno de los nodos de grado máximo. Puede obtenerse una desigualdad más fuerte, considerando la siguiente desigualdad, que se cumple siempre:

$$\sum_{j \in N(i)} c_{ij} \leq d(i) = d(i) \sum_{k \in V(H)} y_{ik}$$

[7] Un grafo es *regular* si todos sus nodos tienen el mismo grado.

Entonces, podemos combinar esta desigualdad con la anterior, obteniendo la siguiente, que la domina:

$$\sum_{j \in N(i)} c_{ij} \leq \sum_{k \in V(H)} min\{d(i), d(k)\} y_{ik}.$$

***

Se presenta ahora otra desigualdad que también corta en algunos casos al punto x^{inf}. Sea $T \subseteq V(G)$. Este conjunto de tareas puede asignarse a cualquier subconjunto de $|T|$ procesadores. Por cada subconjunto W de procesadores, con $|W| = |T|$, se tiene la siguiente desigualdad, que es válida solo para los puntos en los cuales las tareas de T se asignan a los procesadores de W:

$$\sum_{ij \in E(T)} c_{ij} \leq |E(W)| = |E(W)|/|T| \sum_{i \in T} \sum_{k \in W} y_{ik}.$$

Se tiene una desigualdad por cada subconjunto $W \subseteq V(H)$, con $|T| = |W|$, que es válida cuando las tareas de T se asignan a los procesadores de W. Ahora bien, el conjunto de tareas T siempre está asignado a algún subconjunto W (con $|W| = |T|$), con lo cual alguna de estas desigualdades es válida. Por argumentos disyuntivos, podemos tomar los coeficientes mínimos del lado izquierdo y máximos del lado derecho, para obtener una desigualdad que es válida para todo el conjunto de soluciones factibles, con lo cual se tiene la siguiente proposición:

Proposición. *Si $T \subseteq V(G)$, entonces la siguiente desigualdad es válida:*

$$\sum_{ij \in E(T)} c_{ij} \leq |T|^{-1} \sum_{k \in V(H)} w_k y_{ik},$$

donde $w_k := max\{|E(W)| : W \subseteq V(H), |W| = |T|, k \in W\}$.

La siguiente proposición muestra que el cálculo de los coeficientes w_k de esta desigualdad es un problema difícil. Dado un procesador k, el problema de decisión asociado con este cálculo recibe un entero K y consiste en decidir si $w_k \geq K$.

Proposición. *El problema de decisión asociado al coeficiente w_k es NP-completo.*

Demostración. Para probar la proposición, construimos una transformación desde el problema de clique máxima (que recibe un grafo G y un entero K y consiste en decidir si G tiene una clique de peso K o mayor) a este nuevo problema. Dado el grafo G, agregamos un nuevo nodo k y conectamos todos los nodos de G con este nuevo nodo. Esta transformación es polinomial, y cumple que $w_k \geq \frac{K(K+1)}{2}$ si y sólo si existe en G una clique de tamaño K o mayor. Como el problema de clique máxima es NP-Completo, entonces el problema de decisión asociado con w_k también lo es.

Por lo tanto, esta desigualdad es difícil de calcular, aunque en casos particulares se pueden obtener los coeficientes de modo exacto (nuevamente, se pueden reemplazar por cotas superiores). Por ejemplo, si los procesadores están conectados en forma de retícula con cuatro

vecinos por nodo, y tomamos un conjunto T de 9 tareas, se tiene la desigualdad

$$\sum_{ij \in E(T)} c_{ij} \leq \sum_{k \in V(H)} \frac{4}{3} y_{ik}$$

Esta desigualdad no es cumplida por x^{inf} si existe algún subconjunto T con $E(T) > 12$, con lo cual se tiene otra forma de "cortar" a este punto.

Capítulo 6

El algoritmo branch & cut

"El jardín de senderos que se bifurcan es una imagen incompleta, pero no falsa, del universo (...). Esa trama de tiempos que se aproximan, se bifurcan, se cortan o que secularmente se ignoran abarca todas *las posibilidades."*

–Jorge Luis Borges

El estudio poliedral realizado en el capítulo anterior es el punto de partida del algoritmo implementado. Los algoritmos *branch&cut* se basan en el cálculo simultáneo de cotas inferiores y superiores de la solución óptima. Las cotas superiores se obtienen por medio de planos de corte, mientras que las cotas inferiores son halladas por procedimientos heurísticos y durante el proceso de *branching*. Las desigualdades válidas halladas para el problema, junto con procedimientos de separación adecuados, constituyen el núcleo fundamental del algoritmo. De esta forma, las propiedades teóricas del poliedro asociado al problema conforman la piedra angular del método de resolución, tornándose importantes desde el punto de vista práctico.

Por otra parte, para lograr una implementación eficiente se deben considerar muchos otros detalles, que en conjunto aportan a la performance del algoritmo. En este capítulo se describe la implementación, puntualizando sus características específicas. En primer lugar, se enumeran las desigualdades válidas incorporadas al algoritmo y se describen sus procedimientos de separación. En las siguientes secciones, se reseñan las principales características del algoritmo (estrategia de *branching*, uso de heurísticas, etc.), para concluir el capítulo con detalles particulares de la implementación.

6.1 Desigualdades Válidas y Procedimientos de Separación

Esta sección incluye las desigualdades válidas incorporadas al algoritmo, junto con los procedimientos de separación definidos para cada familia de desigualdades. En algunos casos, estas familias están compuestas por una cantidad reducida (lineal o cuadrática) de desigualdades, con lo cual su procedimiento de separación consistirá en chequear cada desigualdad,

para comprobar si es violada en la solución del subproblema. En otros casos, la cantidad de desigualdades es exponencial o polinomial de grado alto, con lo cual se torna ineficiente el chequeo de toda la familia, y es necesaria la definición de algoritmos de separación más complejos, o el uso de técnicas heurísticas.

6.1.1 Familia 1

Las restricciones de costo de la primera formulación considerada para el problema pueden utilizarse como desigualdades válidas de la segunda formulación. Esta es la primera familia utilizada, que está compuesta por las desigualdades

$$c_{ij} + y_{ik} + y_{jl} \leq 2,$$

para $ij \in E(G)$ y $kl \in \bar{E}(H)$. Esta familia está formada por $m_G(\frac{n(n-1)}{2} - m_H) = O(n^2 m_G)$ desigualdades, si $m_G = |E(G)|$ y $m_H = |E(H)|$, con lo cual su separación exhaustiva es computacionalmente inconveniente. En lugar de analizar todas las desigualdades, se utiliza el siguiente procedimiento heurístico sencillo: para cada eje $ij \in E(G)$, se buscan los procesadores $k = \text{argmax}_k y_{ik}$ y $l = \text{argmax}_{l \notin N(k)} y_{jl}$ (aquellos procesadores no conectados que reciban "en mayor medida" las tareas i y j), y se comprueba si la desigualdad formada por el eje ij y los procesadores k y l es violada en el óptimo del subproblema. Si esto sucede, se agrega como un nuevo corte. Este procedimiento de separación tiene un tiempo de ejecución de orden $O(mn)$.

6.1.2 Familia 2

La segunda familia que se incorpora al algoritmo está formada por desigualdades que ayudan a evitar que las tareas se distribuyan en más de un procesador, y se obtuvieron mediante el análisis de las soluciones de la relajación lineal del modelo de programación lineal entera. Dada una tarea $i \in V(G)$ y un subconjunto $W \subseteq V(H)$, entonces

$$\sum_{j \in N(i)} c_{ij} \leq \sum_{k \in N(W)} \sum_{j \in N(i)} y_{jk}$$

es desigualdad válida si $\sum_{k \in W} y_{ik} = 1$ (es decir, si la tarea i se reparte entre los procesadores de W). Se mostró anteriormente (cfr. sección 5.4) la forma de realizar el proceso de *lifting* para llevarla a una nueva desigualdad válida para todo el espacio de soluciones, y estas nuevas desigualdades (luego del proceso de *lifting*) conforman la segunda familia que se utiliza en el algoritmo.

Nuevamente, la cantidad de desigualdades en esta familia es demasiado grande, lo cual impide la implementación de un procedimiento de separación por búsqueda exhaustiva. El procedimiento implementado es la solución "natural" para este problema: Dada la solución $\bar{y}$ del subproblema, para cada tarea i se encuentra el conjunto $W_i = \{k \in V(H) : \bar{y}_{ik} \neq 0\}$, verificando si la desigualdad definida por i y W_i constituye un corte de la solución actual. Debe notarse que otras desigualdades formadas a partir de conjuntos W' distintos también

podrían ser cortes de la solución actual, pero su búsqueda resulta difícil, y el procedimiento de separación implementado se limita a considerar el conjunto W_i para cada tarea i. Con esto, se intenta un balance entre la efectividad de la separación y el tiempo de ejecución de este procedimiento, que en este caso tiene un orden de $O(mn)$.

6.1.3 Familia 3

La familia anterior busca evitar que las tareas se ubiquen en más de un procesador, pero existen extremos de la relajación lineal que no pueden cortar. Entre estos puntos se encuentra el "punto infausto", obtenido al tomar $y_{ik}^{inf} = 1/n$ y $c_{ij}^{inf} = 1$. La tercera familia de desigualdades es reducida, pero corta este punto bajo condiciones generales.

$$\sum_{j \in N(i)} c_{ij} \leq \sum_{k \in V(H)} min\{d(k), d(i)\} y_{ik}$$

Esta familia está formada por n desigualdades, una para cada tarea i. Como se trata de pocas desigualdades, se ubican en el *pool* de cortes al comienzo de la optimización. En cada paso de separación, se intentan regenerar las desigualdades del *pool*, verificando si alguna es violada por la solución del subproblema (cfr. sección 6.1.13). Como estas desigualdades se ubican en el *pool* al comienzo, el proceso de regeneración las agrega automáticamente si son violadas en algún subproblema.

6.1.4 Familia 4

Dado un arco $ij \in E(G)$ del grafo de tareas y un procesador $k \in V(H)$, la siguiente desigualdad es válida:

$$c_{ij} + y_{ik} \leq 2 - \sum_{l \in \bar{N}(k)} y_{jk}.$$

Esta familia está formada por mn desigualdades, con lo cual podría considerarse un procedimiento de separación exhaustiva. Se implementó inicialmente la separación de esta forma, y se observó que la cantidad de cortes producidos por esta familia es muy grande, con lo cual el costo del procedimiento de separación (orden $O(mn^2)$) queda, en cierto modo, justificado por sus resultados.

6.1.5 Familia 5

Si $ij \in E(G)$ es un arco del grafo de tareas y $k \in V(H)$ es un procesador, entonces la siguiente desigualdad es válida:

$$c_{ij} \leq \sum_{l \in N(k)} y_{jl} + \sum_{l \neq k} \psi'_l y_{il} + \sum_{j \neq i} y_{jk}.$$

Esta desigualdad surge de considerar $c_{ij} \leq \sum_{l \in N(k)} y_{jl}$, que es válida si $y_{ik} = 1$, realizando luego *lifting* sobre las variables y_{il} $(l \neq k)$ e y_{jk} $(j \neq i)$. Los valores ψ'_l corresponden a los

coeficientes de *lifting*, mientras que los coeficientes de *lifting* para las variables y_{jk} son $\psi_j = 1$, de acuerdo a la descripción dada en la sección 5.3.

La separación de estas desigualdades se realiza con un procedimiento heurístico sencillo. Para cada eje $ij \in E(G)$, se busca el procesador $k = \text{argmax}_k y_{ik}$, y se verifica si la desigualdad definida por ij y k es violada en el subproblema actual. Al buscar el procesador con mayor valor y_{ik}, se espera que el lado derecho de la desigualdad tenga un valor pequeño, acotando a la variable c_{ij} si ésta toma un valor mayor que el que debería tener, producto de las restricciones con variables fraccionarias.

Para verificar si la desigualdad es violada, se consideran los coeficientes de *lifting* como $\psi'_l = -1$. No es difícil verificar que $\psi'_l \geq -1$, con lo cual pueden reemplazarse estos coeficientes por su cota superior manteniendo la validez de la desigualdad. Con esto, se pueden agregar desigualdades que no corten a la solución actual. Estas desigualdades se incorporan al subproblema y no tienen incidencia en la siguiente iteración, pero quedan en el *pool* de cortes, y pueden ser regeneradas en iteraciones subsiguientes.

Debe notarse que los coeficientes de *lifting* se consideran con valor $\psi'_l = -1$ sólo para verificar si la desigualdad es violada en el óptimo del subproblema. Sin embargo, si se encuentra un corte, se calculan efectivamente los coeficientes exactos y la desigualdad se agrega a la formulación con estos coeficientes.

6.1.6 Familia 6

Dada una tarea $i \in V(G)$ y un procesador $k \in V(H)$, consideremos la siguiente desigualdad (cfr. sección 5.3):

$$\sum_{j \in N(i)} c_{ij} \leq d(k) - \sum_{l \notin N(k)} \sum_{j \in N(i)} y_{jl} \quad \text{si } y_{ik} = 1.$$

Nuevamente, se recurre al *lifting* para obtener desigualdades válidas para todo el espacio de soluciones, que son las que conforman efectivamente esta familia. El procedimiento de separación busca para cada tarea i el procesador k con mayor valor y_{ik}, y verifica si la desigualdad definida es violada en la solución del subproblema. Para realizar la comprobación, los coeficientes de *lifting* se consideran nulos, con lo cual se pueden agregar desigualdades que no corten la solución actual, pero que pueden ser útiles en iteraciones posteriores. Del mismo modo que en la familia anterior, cuando se encuentra un corte se calculan los valores exactos de los coeficientes de *lifting*.

6.1.7 Familia 7

Sea $I \subseteq V(H)$ un conjunto independiente de procesadores. Entonces, la siguiente desigualdad es válida para cualquier eje $ij \in E(G)$:

$$c_{ij} + \sum_{k \in I} (y_{ik} + y_{jk}) \leq 2$$

Con estas desigualdades, se lleva a cabo otro intento para acotar los valores de las variables de costo c_{ij}. Esta familia está formada por un número exponencial de desigualdades, con lo cual su procedimiento de separación debe ser, nuevamente, más refinado que una búsqueda exhaustiva entre todas ellas. El procedimiento implementado considera cada eje $ij \in E(G)$ e intenta hallar un conjunto independiente I de modo tal que la desigualdad definida por el conjunto esté violada en el óptimo del subproblema. Dado un eje ij, se busca un conjunto independiente I en forma golosa, con el siguiente procedimiento:

```
I <- conjuntoVacio();
repetir
    Buscar k que no este en I, tal que I+{k} sea
    independiente y que maximice y[i][k] + y[j][l];

    I <- I+{k};
fin(repetir);
```

Para cada eje ij, se busca un conjunto independiente con este procedimiento goloso y se verifica si la desigualdad definida por el eje y el conjunto es violada por el óptimo del subproblema. Si esto sucede, se agrega el corte. Este procedimiento de separación tiene un tiempo de ejecución de orden $O(mnk)$, si k es el cardinal del mayor conjunto independiente del grafo de procesadores.

6.1.8 Familia 8

Dados t conjuntos independientes $I_1, I_2, ..., I_t$ del grafo de procesadores, llamemos $c(k)$ a la cantidad de conjuntos que incluyen al procesador k. Dado un eje $ij \in E(G)$, se tienen las siguientes desigualdades (cfr. sección 5.4):

$$c_{ij} + \sum_{k \in V(H)} \lfloor c(k)/2 \rfloor (y_{ik} + y_{jk}) \leq \lfloor t/2 \rfloor .$$

Estas desigualdades son redundantes cuando t es par, pero constituyen cortes cuando t es impar. El procedimiento de separación de estas desigualdades presenta algunos problemas. En primer lugar, se debe definir si se fija un valor t de antemano o se permite cualquier número de conjuntos independientes. Por otra parte, deben hallarse efectivamente estos conjuntos de modo que la desigualdad definida corte a la solución actual. Por último, el procedimiento de separación debe ser eficiente, puesto que de otro modo incidiría negativamente en la performance total.

No es difícil verificar que estas desigualdades se reducen a la familia anterior cuando $t = 3$. Por otra parte, considerar $t \geq 7$ puede originar costos demasiado elevados, con lo cual se fijó $t = 5$ para la separación. El procedimiento es heurístico, y construye de a uno los conjuntos independientes, del siguiente modo:

```
c[k] <- 0, para todo k
Para t desde 1 hasta 5:

    It <- conjuntoVacio();
    Repetir hasta que It sea independiente maximal:

        Buscar el procesador k independiente de It con mayor valor de
        0.5 * (1+c[k]) * ( y[i][k] + y[j][k] ) y agregarlo a It.

        c[k] <- c[k] + 1;

    Fin (repetir)

Fin (para)
```

Dado que este procedimiento de separación puede ser muy costoso, se realiza solamente con los ejes ij que tengan $\bar{c}_{ij} > 0.5$ en la solución actual. Este procedimiento es heurístico, y puede suceder que no sea suficiente para detectar cortes de esta familia, o que las desigualdades violadas no tengan la forma particular ($t = 5$) que se adoptó para la separación. Se realizaron pruebas computacionales para verificar estos aspectos de la separación, que se describen en la sección 7.2.2.

6.1.9 Familia 9

Esta familia es sencilla y está compuesta por las desigualdades

$$\sum_{j \in N(i)} c_{ij} \leq \max\{d(k) : k \in V(H)\},$$

para cada tarea $i \in V(G)$. Las n desigualdades de la familia pueden separarse por inspección exhaustiva, con lo cual se ubican en el *pool* de cortes al comienzo del proceso, de modo tal que el procedimiento de regeneración de cortes (descripto más adelante) se encargue de su separación.

6.1.10 Familia 10

Sean i una tarea, k un procesador, y $T \subseteq N(i)$ un conjunto de vecinos de i. La familia 10 está compuesta por las siguientes desigualdades:

$$\sum_{j \in T} c_{ij} \leq \sum_{j \in T} \sum_{l \in N(k)} y_{jl} + |T|(1 - y_{ik}).$$

Para cada tarea i, se tienen $2^{d(i)-1}$ sobconjuntos $T \subseteq N(i)$ no vacíos, con lo cual la familia está formada por $\sum_i 2^{d(i)-1}$ desigualdades. Dada una tarea i, la inspección exhaustiva de todos los subconjuntos $T \subseteq N(i)$ es muy costosa si la tarea tiene grado alto. Sin embargo,

puede desarrollarse un procedimiento de separación polinomial que encuentre al menos una desigualdad violada si existe alguna en estas condiciones. Una vez fijada la tarea i y un procesador k, debemos hallar un subconjunto T de vecinos de la tarea que maximice la expresión

$$w = \max_{T \subseteq N(i)} \sum_{j \in T} \Big(\bar{c}_{ij} - \sum_{l \in N(k)} \bar{y}_{jl} \Big) - |T|(1 - \bar{y}_{ik}).$$

Si $w > 0$, entonces el conjunto T que lo realiza define una desigualdad violada de esta familia. Si $w \leq 0$, entonces ningún subconjunto de vecinos define una desigualdad violada. Podemos introducir una variable binaria x_j para cada $j \in N(i)$, para escribir este máximo como un problema lineal:

$$w = \max_{x_j \in \{0,1\}} \sum_{j \in N(i)} \Big(\bar{c}_{ij} - \sum_{i \in N(k)} \bar{y}_{jl} \Big) x_j - (1 - \bar{y}_{ik}) \sum_{j \in N(i)} x_j.$$

En este problema, el término $\sum x_j$ indica el cardinal del conjunto. Los vecinos j con $x_j = 1$ se encuentran dentro del conjunto T, que define una desigualdad violada si $w > 0$. Ahora bien, si llamamos

$$d_j := \bar{c}_{ij} - \sum_{l \in N(k)} \bar{y}_{jl} + \bar{y}_{ik} - 1,$$

puede verse que en este último problema se maximiza $\sum d_j x_j$, sin restricciones sobre las variables. El óptimo se obtiene tomando $x_j = 1$ si $d_j \geq 0$ y $x_j = 0$ en caso contrario. Entonces, el procedimiento de separación para esta familia de desigualdades considera a todas las tareas j vecinas de i, agregando al conjunto T aquellas con $d_j \geq 0$. Este procedimiento se repite para cada tarea i y cada procesador k, con lo cual tiene un tiempo de ejecución polinomial. Sin embargo, el tiempo puede ser muy alto al considerar todos los procesadores k, con lo cual se tiene en cuenta sólo el procesador k con mayor valor de $\bar{y}_{ik}$. El siguiente pseudocódigo resume el procedimiento:

```
Para cada tarea i:

    k <- argmax y[i][k];
    T <- conjuntoVacio();
    w <- 0.0;

    Para cada vecino j de la tarea i, con d[j] > 0:
        T <- T + {j};
        w <- w + d[j];
    Fin (para)

Fin (para)
```

6.1.11 Familia 11

Esta es una de las familias exponenciales de facetas. Las desigualdades que la componen están definidas por un eje ij del grafo de tareas y un subconjunto W de procesadores (cfr.

sección 5.4).

$$c_{ij} \leq \sum_{l \in N(W)} y_{jl} + \sum_{k \notin W} \psi_k y_{ik}.$$

Los coeficientes ψ_k provienen del proceso de *lifting*, pero no es necesaria su obtención en el procedimiento de separación. Para cada tarea i, se considera el conjunto $W = \{k : y_{ik} \neq 0\}$, y se verifica si la desigualdad definida por i y W es violada por el óptimo del nodo. Este procedimiento heurístico proporciona buenos resultados, que se resumen en la sección 7.2.2.

6.1.12 Familia 12

Al igual que la anterior, esta familia tiene un número exponencial de desigualdades, que definen facetas del poliedro. Dado un eje ij del grafo de tareas y un conjunto W de procesadores, esta familia está formada por las siguientes desigualdades válidas (cfr. sección 5.4):

$$c_{ij} + \sum_{k \in W} y_{ik} \leq 1 + \sum_{l \in N(w)} y_{jl}.$$

El procedimiento de separación es similar al anterior, en el cual se busca el conjunto de procesadores W que cumplan $y_{ik} \neq 0$, y se verifica si la desigualdad resultante está violada. Si esto sucede, se agrega como un nuevo corte. Este procedimiento es heurístico, pero tiene un tiempo de ejecución polinomial de grado bajo, lo cual es importante dado que se debe ejecutar muchas veces durante todo el proceso.

6.1.13 Manejo de los cortes

Luego de resolver la relajación lineal de la formulación de cada nodo, y si la solución obtenida no es factible entera, se ejecuta el procedimiento de separación para generar desigualdades que corten el óptimo de esta relajación. Los cortes generados se agregan al subproblema, que se vuelve a resolver. Estos pasos conforman una *iteración*, que se repite un cierto número de veces, determinado por los parámetros del algoritmo. Cuando se decide realizar un paso de *branching*, los nuevos nodos, hijos del actual, heredan el sistema de restricciones final de su padre en el árbol de enumeración. Es decir, las restricciones de los nuevos nodos serán las mismas que tuvo su padre en su último problema lineal, junto con una restricción adicional, debida al proceso de *branching*.

Cada nuevo corte que se genera durante los procedimientos de separación se guarda en un *pool* de cortes (cfr. sección 6.5.2), por dos motivos. En primer lugar, se ahorra memoria dado que una misma desigualdad puede ser utilizada en más de un nodo del árbol de enumeración. Por otra parte, se tiene un procedimiento adicional de separación: antes de ejecutar los procedimientos de separación descriptos, se revisa este *pool* de desigualdades, para verificar si alguna de ellas es violada en el óptimo actual. Si esto sucede, se agrega como un nuevo corte.

Este proceso se denomina *regeneración de cortes*, y tiene tres ventajas importantes (cfr. [Jun/98]). En primer lugar, la separación directa del *pool* puede ser más rápida que la ejecución de los procedimientos implementados. Por otra parte, si una clase de desigualdades sólo puede

ser separada por métodos heurísticos, puede suceder que el procedimiento no la pueda generar, pero si esta desigualdad ya apareció anteriormente será regenerada fácilmente desde el *pool*. Por último, este proceso produce una gran cantidad de cortes por cada nodo, como se verá más adelante.

Por lo tanto, el procedimiento general de separación consta de dos etapas. En primer lugar, se verifica si alguna desigualdad del *pool* de cortes es violada en el óptimo actual (regeneración de cortes), y se agregan como cortes aquellas que lo sean. En segundo lugar, se ejecutan los procedimientos de separación para cada una de las familias, de acuerdo con las descripciones correspondientes. Se agregan a la formulación del subproblema los cortes provinientes de ambas etapas.

6.2 Heurísticas primales

Durante las pruebas realizadas con una implementación preliminar del algoritmo, se observó que resulta dificultoso hallar soluciones factibles del problema entero por medio del procedimiento de *branching*. Este fenómeno en el comportamiento del algoritmo se repitió al reimplementar el algoritmo utilizando una versión actualizada de ABACUS, y se observó un comportamiento similar con el algoritmo *branch&bound* implementado por CPLEX. Por estos motivos, se hizo necesario el agregado de técnicas para subsanar esta situación, que se orientaron en dos direcciones: (a) Construcción de una solución factible inicial, y (b) Aprovechamiento de las soluciones de los subproblemas lineales para construir soluciones factibles.

6.2.1 Solución factible inicial

Como se observó la dificultad que tenía el proceso de *branching* para hallar soluciones factibles, se decidió implementar una heurística "agresiva", permitiendo que utilice una porción importante de tiempo de ejecución. De esta forma, se espera que esta heurística inicial arroje una buena solución factible, que proporcione una cota inferior importante del óptimo. La heurística implementada es una adaptación de la técnica metaheurística *Chained Local Optimization* (CLO), introducida en [Mar/96], tal como fue descripta en la sección 3.3. Esta técnica combina las ideas de *simulated annealing* con procedimientos de búsqueda local, y ha arrojado muy buenos resultados para este problema.

Las pruebas realizadas con este método permitieron contar con rangos apropiados para los parámetros. Con el objetivo de tener un procedimiento suficientemente robusto, al cual se le permita ejecutar durante un tiempo razonable (pero no excesivo), se fijaron los parámetros de esta heurística del siguiente modo: La temperatura inicial se calcula en forma automática, de acuerdo al procedimiento descripto, y se reduce en un factor de 0.95 luego de cada época, hasta llegar a una temperatura mínima de 0.01, o hasta que hayan transcurrido 5 épocas consecutivas sin mejorar la solución. Se realizan $30\lceil\frac{n}{50}\rceil$ iteraciones por época, siendo n la cantidad de nodos del problema.

6.2.2 Heurística de mejoramiento primal

La resolución de los subproblemas del árbol de enumeración no produce, en general, soluciones factibles. Sin embargo, puede suceder que estos puntos contengan información estructural que permita obtener buenas soluciones factibles del problema entero (cfr. [Jun/94], pág. 25). La construcción de una solución entera a partir del resultado de la optimización del subproblema puede conducir a soluciones mejores que las consideradas hasta el momento, posibilitando la reducción de la garantía de optimalidad e inclusive la obtención del óptimo global. Se implementó una heurística de redondeo que permite obtener una asignación factible a partir del óptimo de un subproblema, y que se describe en esta sección.

La heurística implementada consta de dos fases: en la primera se construye por redondeo una solución factible a partir del óptimo del subproblema (que es de naturaleza lineal), y en la segunda fase se aplica un procedimiento heurístico para mejorar la solución obtenida. Se utiliza como procedimiento heurístico el algoritmo de rebotes simulados, tomando como punto de partida la solución hallada en la primera fase.

La construcción de la *primera fase* procede ubicando cada tarea sobre el procesador cuya variable de asignación sea mayor. Es decir, la tarea i se asigna al procesador $\text{argmin}_k\ y_{ik}$. Puede suceder que este proceso asigne más de una tarea a un mismo procesador (con lo cual quedan procesadores sin recibir tareas). Para obtener una solución factible, se reparten aleatoriamente las tareas "sobrantes" de cada procesador entre los procesadores libres. Con esto, se completa la construcción por redondeo de una solución factible.

Puede analizarse la calidad de las soluciones obtenidas con este método, considerando la distribución de valores de la función objetivo. El gráfico 6.1 resume los valores de la función objetivo logrados por este procedimiento de redondeo (línea de puntos), que se comparan con los valores obtenidos generando soluciones aleatoriamente (línea contínua). Este gráfico muestra la frecuencia de aparición de cada valor de la función objetivo, es decir, cuántas veces se obtuvo cada valor con cada procedimiento. Puede verse que la calidad de las soluciones del procedimiento de redondeo no difiere en gran medida de la calidad de las soluciones generadas al azar, con lo cual este procedimiento no aporta, en principio, una mejora sustancial, y se torna necesario un ajuste.

El procedimiento mejorado de redondeo modifica la asignación de las tareas "sobrantes" (aquellas que se asignan a procesadores ocupados). La asignación de una de estas tareas a un procesador libre hace que la función objetivo de la solución parcial se mantenga o mejore. En este nuevo procedimiento, se asignan secuencialmente las tareas sobrantes al procesador que maximiza la mejora de la función objetivo. Es decir, la tarea i se asigna al procesador $\text{argmax}_k\ \#\{j \in N(i) : y_{il} = 1, \text{para algún } l \in N(k)\}$. Se repitieron las pruebas con este nuevo procedimiento de redondeo, obteniendose los resultados que muestra el gráfico 6.2.

Como puede verse, la calidad de las soluciones obtenidas con este segundo procedimiento de redondeo es superior, con lo cual se optó por este método para la implementación del algoritmo *branch&cut*. Esta pequeña modificación hizo que el tiempo de resolución para el problema STR5 disminuyera (en una implementación preliminar) de 44:58 a 1:12 minutos.

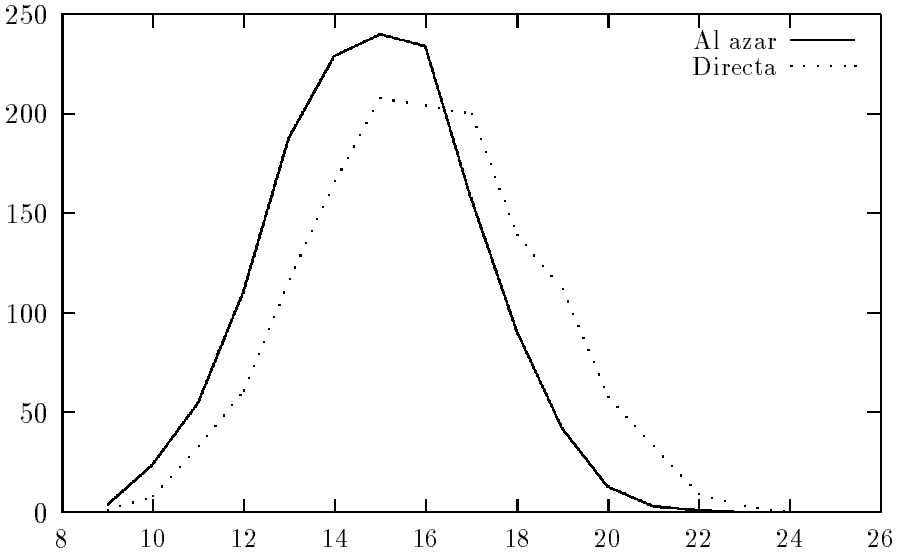


Figura 6.1: Soluciones (a) aleatorias y (b) generadas por el método de redondeo (cantidad de veces que se halló cada valor de la función objetivo)

En la *segunda fase* de la heurística primal, se ejecuta el algoritmo de rebotes simulados a partir de la solución hallada en la primera fase. Los parámetros utilizados varían de acuerdo a la relación entre la solución s hallada por redondeo y la mejor solución s^* hallada hasta el momento. Si $f(s) \geq 0.7f(s^*)$, se utilizan parámetros que hacen que la búsqueda sea más intensiva, dado que se parte de una solución aparentemente "prometedora": en este caso, se fijan 400 iteraciones con $\alpha = 0.999$, y en caso contrario se realizan 100 iteraciones con $\alpha = 0.99$.

6.3 Estrategia de *branching*

El procedimiento de *branching* es uno de los puntos más importantes del algoritmo. En este paso, se divide el subproblema en dos o más nuevos subproblemas hijos, representados por nuevos nodos en el árbol de enumeración. Los nuevos nodos pasan a la lista de subproblemas activos, mientras que su padre pasa a la lista de nodos no activos. Este procedimiento define efectivamente la forma que tendrá el árbol de enumeración, y tiene gran incidencia en la performance del proceso total. El modo en el que se realiza esta división da lugar a diferentes estrategias de *branching*.

Una de las estrategias más utilizadas consiste en generar dos nuevos subproblemas separando el rango de valores de una cierta variable x_i. El primer subproblema tendrá como soluciones factibles aquellas soluciones x de su padre con $x_i \leq k$, mientras que el segundo retendrá las soluciones con $x_i \geq k + 1$. Se realiza este paso si $k < \bar{x}_i < k + 1$ en la solución actual, con lo cual las soluciones óptimas de los nuevos subproblemas no coincidirán con la solución óptima de su padre, permitiendo así que la búsqueda continúe por nuevos puntos

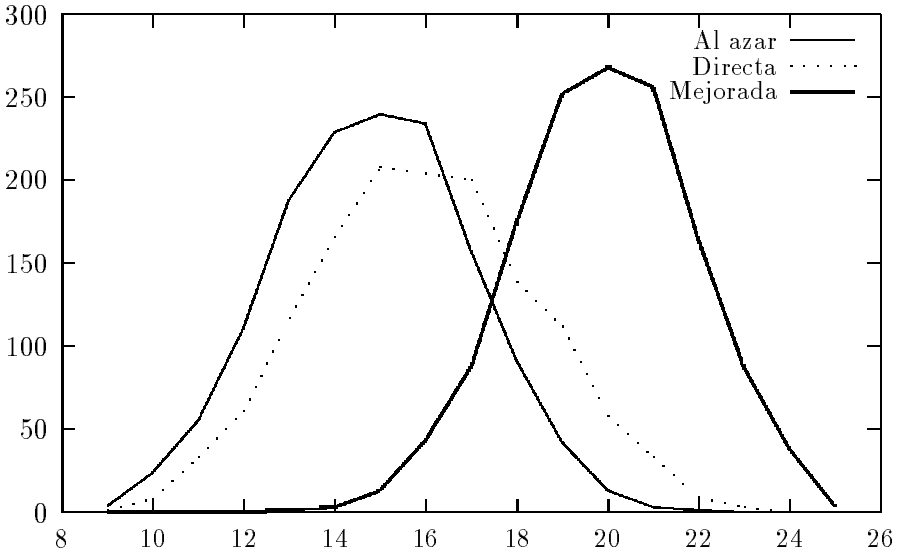


Figura 6.2: Soluciones generadas por el método de redondeo mejorado (cantidad de veces que se halló cada valor de la función objetivo)

del espacio de soluciones. En problemas binarios, esta estrategia se especializa generando dos subproblemas con $x_i = 0$ y $x_i = 1$, respectivamente. Existen distintas posibilidades para seleccionar la variable x_i sobre la cual se realiza el proceso (cfr. [Jun/94]).

Esta estrategia de *branching* es muy utilizada, pero la forma particular de las restricciones del problema de *mapping* permite considerar otro modo de realizar este proceso (cfr. [Joh/98]). En esta nueva estrategia, se asocia a cada tarea de cada subproblema un rango de procesadores entre los cuales se puede ubicar. Si el rango de la tarea i es $(k_i, k'_i - 1)$, entonces el subproblema tendrá asociadas las restricciones

$$\sum_{l=k_i}^{k'_i-1} y_{il} = 1, \qquad i = 1, ..., n,$$

de modo tal que la tarea i sólo puede asignarse a los procesadores $k_i, k_i + 1, ..., k'_i - 1$. Inicialmente, todas las tareas tienen rango $(0, n)$. En el paso de *branching*, se selecciona la tarea i que tenga mayor rango (k_i, k'_i) y que se pueda dividir por la mitad (es decir, si $t = k'_i - k_i$ debe cumplirse que $0 < \sum_{l=k_i}^{k_i+t/2} \bar{y}_{il} < 1$). Si esta tarea existe, se generan dos nuevos subproblemas, agregando las restricciones

$$\sum_{l=k_i}^{k_i+\lfloor t/2 \rfloor} y_{il} = 1 \qquad\qquad \sum_{l=k_i+\lfloor t/2 \rfloor+1}^{k'_i} y_{il} = 1.$$

Si ninguna tarea tiene un rango que se pueda dividir por la mitad, se recurre al *branching* standard por variables, seleccionando una variable que se fija en 0 en el primer subproblema y en 1 en el segundo[1].

[1] Otra posibilidad podría ser dividir el rango de modo tal que cada mitad tenga un costo aproximadamente igual a 1/2, y es de interés analizar el comportamiento del algoritmo en este caso.

6.3.1 Detección de subproblemas no factibles

A medida que se avanza en el proceso de enumeración, el proceso de *branching* puede introducir condiciones inconsistentes, que hacen que el subproblema entero resulte no factible. Si un nodo del árbol de enumeración tiene asociado un subproblema que no es factible, entonces se puede cerrar este nodo, de modo tal que la detección de infactibilidad de un subproblema se torna un elemento importante que contribuye a la eficiencia del algoritmo. En esta sección se encuentran las causas que pueden hacer que un subproblema resulte no factible, y se prueba que esta situación es siempre detectada por el proceso de resolución de su problema lineal asociado. A lo largo de esta sección, se asume que solamente se utiliza la regla de *branching* descripta anteriormente, sin recurrir al *branching* por variables.

La regla de *branching* utilizada reduce el rango sobre el cual pueden asignarse las tareas, de modo tal que un subproblema está determinado por los rangos permitidos de cada tarea. Supongamos que los procesadores están indexados entre 1 y n, es decir, $V(H) = \{1, ..., n\}$, y consideremos un subproblema S con rangos $R_i = [m_i, M_i]$ asociados a cada tarea i ($0 \leq m_i, M_i \leq n$). El siguiente teorema caracteriza las razones que hacen que el problema entero asociado con el subproblema sea no factible.

Teorema. *El subproblema entero S es no factible si y sólo si existe un rango de procesadores $P = [p, p']$ ($0 \leq p, p' \leq n$) tal que $\#\{i : R_i \subseteq P\} > |P|$.*

Demostración. $\Leftarrow$) Si $P = [p, p']$ y existen más de $|P|$ tareas cuyos rangos están incluidos en P, entonces se tiene una cantidad mayor que $|P|$ tareas para asignarse en $|P|$ procesadores, lo cual no es posible. Luego, el subproblema S no es factible.

$\Rightarrow$) Consideremos la afirmación contrarrecíproca, es decir, si $\#\{i : R_i \subseteq P\} \leq |P|$ para todo rango $P = [p, p']$ entonces el subproblema es factible. Se muestra cómo construir una solución factible a partir de estas condiciones, con lo cual S es factible. El siguiente procedimiento construye una solución factible:

1. $A \leftarrow \emptyset$ (conjunto de tareas asignadas)
2. Para k desde 1 hasta n:
   (a) Sea $T = \{i : i \notin A \text{ y } k \in R_i\}$.
   (b) Sea i la tarea de T con menor M_i.
   (c) Asignar la tarea i al procesador k.
   (d) $A \leftarrow A \cup \{i\}$
3. Fin

Este procedimiento construye una solución factible del subproblema (asignando cada tarea i a un procesador de R_i), si en cada paso k es posible hallar una tarea i para asignarse al procesador k. Para completar la demostración, sólo resta ver que siempre se puede hallar una

tarea en estas condiciones, es decir, que el conjunto T definido en el paso 2(a) nunca es vacío. Supongamos, entonces, que estamos en el paso k, en el cual los procesadores $1, ..., k-1$ ya recibieron las tareas del conjunto A.

Supongamos que T es vacío, es decir, que ninguna tarea se puede asignar en el procesador k. Esto sucede si ninguna tarea fuera de A tiene rango que incluya al procesador k. Se presentan dos casos:

- $M_i < k$ para todas las tareas $i \in A$. Entonces, o bien (a) existe alguna tarea $i_0 \notin A$ con $M_{i_0} < k$, o bien (b) todas las tareas $i \notin A$ tienen $k < m_i$. En el caso (a), el rango $[1, k-1]$ tiene longitud $k-1$ e incluye los rangos de k tareas (las tareas de A y la tarea i_0), lo cual no puede suceder. En el caso (b), el rango $[k+1, n]$ tiene longitud $n-k$ e incluye los rangos de $n-k+1$ tareas, contradiciendo las hipótesis.
- Si existe alguna tarea $i \in A$ con $M_i \geq k$, entonces como $T = \emptyset$, $m_i > k$ para todas las tareas $i \notin A$. Nuevamente, el rango $[k+1, n]$ incluye los rangos de $n-k+1$ tareas, lo cual no puede suceder.

En ambos casos, se llega a un absurdo, que proviene de suponer que no existe ninguna tarea posible para asignar al procesador k. Por lo tanto, cada paso del ciclo se puede llevar a cabo, con lo cual el procedimiento anterior construye una solución factible. Entonces, S es factible.

Este teorema caracteriza los subproblemas S que no tienen soluciones enteras, pero puede suceder que la relajación lineal de S (que es la que se resuelve efectivamente cuando se procesa este nodo) sea factible. La siguiente proposición muestra que si el problema entero asociado con S es no factible, entonces su relajación lineal también lo será. De este modo, se puede detectar la no factibilidad de un subproblema considerando solamente la factibilidad de su relajación lineal.

Proposición. *Sea S un subproblema. Entonces S es no factible si y sólo si su relajación lineal es no factible.*

Demostración. Si la relajación lineal no es factible, entonces tampoco lo es el problema entero, con lo cual sólo se debe probar la afirmación recíproca. Sea entonces S no factible y supongamos que su relajación lineal es factible. Sea $(\bar{y}, \bar{x})$ una solución de esta relajación lineal. Como S es no factible, por el teorema anterior existe un rango P de procesadores tal que $T = \{i : R_i \subseteq P\}$ cumple $|T| > |P|$. Sean $\bar{P} = \{k : k \notin P\}$ y $\bar{T} = \{k : k \notin T\}$.

Como $\bar{T}$ tiene $n - |T|$ tareas, entonces

$$\sum_{i \in \bar{T}} \sum_{k \in \bar{P}} y_{ik} \leq \sum_{i \in \bar{T}} \sum_{k} y_{ik} = n - |T|.$$

Por otra parte, $\bar{P}$ tiene $n - |P|$ procesadores, y como las tareas de T sólo pueden asignarse a procesadores de P, se tiene que $y_{ik} = 0$ para $i \in T$ y $k \in \bar{P}$, con lo cual

$$\sum_{i \in \bar{T}} \sum_{k \in \bar{P}} y_{ik} = \sum_{i} \sum_{k \in \bar{P}} y_{ik} = n - |P|.$$

Entonces, se tiene que

$$n - |P| = \sum_{i \in \bar{T}} \sum_{k \in \bar{P}} y_{ik} \leq n - |T|,$$

es decir, $|T| \leq |P|$, contradiciendo la hipótesis del absurdo. La contradicción proviene de suponer que la relajación lineal de S es factible, con lo cual su relajación lineal no es factible.

Una de las razones que permiten cerrar un nodo es la detección de no factibilidad del problema entero asociado. Como corolario de esta proposición, no es necesario implementar procedimientos para verificar la factibilidad de los subproblemas enteros, dado que este chequeo es realizado implícitamente por las rutinas que resuelven su relajación lineal. Si la relajación lineal de un subproblema es no factible, entonces automáticamente se cierra el nodo, y en virtud de la proposición anterior, esto sucede cuando el subproblema entero no tiene soluciones factibles[2].

6.4 Fijado de variables por implicaciones lógicas

Durante el proceso de *branching* se limitan los valores que pueden tomar las variables, y esta información puede utilizarse para fijar los valores de algunas de ellas. De esta forma, se reduce el tamaño de los problemas lineales a resolver, mejorando la performance del proceso. El algoritmo implementado hace uso de esta situación, fijando las variables de costo c_{ij} cuando es posible.

La variable c_{ij} puede fijarse en 0 cuando las tareas i y j están obligadas a ubicarse en procesadores que no pueden ser adyacentes. Más precisamente, se fija $c_{ij} = 0$ cuando los rangos de las tareas i y j son $R_i \subseteq V(H)$ y $R_j \subseteq V(H)$ respectivamente, y no existe ningún eje $kl \in E(H)$ con $k \in R_i$ y $l \in R_j$. Los rangos de las tareas están determinados por el proceso de *branching*, y cada subproblema conoce el rango de procesadores sobre el cual puede asignarse cada tarea.

Debe notarse que no es necesario implementar un procedimiento para fijar las variables c_{ij} en 1. Esto es posible si $kl \in E(H)$ para todo $k \in R_i$ y todo $l \in R_j$, pero en este caso las restricciones sobre las variables de costo se limitan a afirmar que $c_{ij} \leq 1$, y como esta variable aparece con coeficiente positivo en la función objetivo (y su valor es independiente de las otras variables de costo), entonces tomará valor $\bar{c}_{ij} = 1$ en el óptimo del subproblema.

[2]Debe mencionarse la posibilidad de desarrollar algoritmos específicos para detectar la factibilidad del subproblema entero, basados en los resultados de esta subsección, que se ejecuten antes de resolver la relajación lineal.

6.5 Otros detalles

6.5.1 Formulación inicial

La formulación con la que se comienza el proceso de optimización está formada por las restricciones de asignación y de costo del segundo modelo (que conforman la formulación original del problema). Con esto, se tienen $O(mn)$ restricciones en el modelo inicial. La cantidad de filas de la formulación inicial posibilita que sus restricciones se consideren estáticas (se mantienen a lo largo de todo el proceso de optimización). Con esto, el chequeo de factibilidad de la solución de un subproblema se limita a verificar si sus variables toman valores enteros.

6.5.2 *Pools* de Desigualdades

El algoritmo mantiene tres *pools* de desigualdades, de acuerdo con los requerimientos de ABACUS. De esta forma, se organiza el almacenamiento de las restricciones y de los cortes que surgen durante su ejecución.

Pool de restricciones. Este *pool* guarda las restricciones que forman parte de la formulación inicial. Estas restricciones son estáticas (están presentes en todos los subproblemas), y se ubican al comienzo de la ejecución. Este *pool* no se modifica durante el desarrollo del algoritmo, y está predefinido en la implementación de ABACUS.

Pool de cortes. Aquí se guardan las desigualdades válidas que surgen de los procedimientos de separación a lo largo del algoritmo. Este *pool* contiene todas las desigualdades, que se guardan durante el proceso desde el momento en el que ingresan. Luego de la resolución de cada subproblema lineal, se recorren las desigualdades de este *pool*, para verificar si alguna de ellas es violada por el óptimo del subproblema. Al igual que el anterior, este *pool* es predefinido de ABACUS.

Como los procedimientos de separación se llaman para todos los subproblemas, independientemente de si se pudieron regenerar desigualdades del *pool*, es posible que contenga elementos duplicados. Sin embargo, su detección es costosa, con lo cual se adoptó esta solución, aunque produzca entradas duplicadas. Para evitar esta situación podrían no aplicarse los procedimientos de separación si se regeneraron cortes del *pool*, pero esta opción no produce buenos resultados (cfr. sección 7.3.5).

Pool de branching. Cuando se decide realizar *branching* en un subproblema (utilizando la estrategia refinada), se genera una restricción para cada subproblema hijo. Estas dos nuevas restricciones se agregan a un *pool* especial, que inicialmente se fija con espacio para 5000 restricciones. Con esto, se pueden tener inicialmente hasta 5000 subproblemas. De acuerdo con las pruebas realizadas, este valor es adecuado para el límite de tiempo utilizado, pero debería aumentarse si se utilizan límites mayores. Cuando el espacio en este *pool* se agota, es redimensionado automáticamente para permitir que ingresen nuevas desigualdades. Sin embargo, no es conveniente que esto suceda con frecuencia, dado que la performance se ve afectada.

Capítulo 7

Resultados computacionales

"Es evidente que nos precipitamos hacia algún apasionante descubrimiento, un secreto incomunicable cuyo conocimiento entraña la destrucción."

–Edgar Allan Poe

El algoritmo general *branch&cut* tiene garantizada su terminación en tiempo finito, dado que el esquema de enumeración finaliza a lo sumo luego de haber recorrido todas las soluciones factibles (sobre un dominio finito). Sin embargo, no existe garantía polinomial sobre el tiempo de ejecución en el peor caso, que puede exceder todas las previsiones dada la explosión exponencial que puede verificar el crecimiento del árbol de enumeración. Una característica importante de este tipo de algoritmos es que su tiempo de ejecución tiende a mejorar a medida que se conoce en mayor medida la estructura del poliedro asociado al problema, que se traduce en nuevas y más fuertes desigualdades válidas. De esta forma, la experimentación computacional permite conocer la fuerza empírica de las desigualdades incorporadas al método, y constituye una importante herramienta para evaluar la performance del algoritmo, además de ayudar en el ajuste de sus parámetros.

Las pruebas realizadas se dividen en tres grandes grupos. En primer lugar, se llevaron a cabo experimentos computacionales con las desigualdades válidas, para obtener evidencia de su calidad, que se describen en la sección 7.2. Por su parte, la sección 7.3 resume las pruebas orientadas a conocer el comportamiento del algoritmo *branch&cut* al variar los valores de sus parámetros, entre los cuales se encuentran las estrategias de recorrido del árbol, estrategias de *branching*, generación de cortes, etc. Se realizaron comparaciones entre distintos métodos, y contra el paquete CPLEX de optimización lineal y lineal entera, que se detallan en la sección 7.4. Por último, la sección 7.6 presenta extensivamente los resultados obtenidos sobre todas las instancias consideradas.

7.1 Instancias de prueba

Se utilizaron varios grupos de instancias para realizar las pruebas computacionales. Las instancias provienen de distintas fuentes, entre las que se encuentran (a) instancias del problema de *mapping* simple, (b) instancias de otras variaciones del problema de *mapping*, (c) instancias adaptadas de otros problemas de *scheduling*, y (d) problemas generados al azar. Excepto el último grupo, todas las instancias están tomadas de artículos y archivos de problemas.

Arquitecturas paralelas. En muchos artículos se consideran arquitecturas en grilla, con 4, 6 u 8 vecinos por nodo (fig. 7.1). Normalmente, los procesadores ubicados "en los bordes" de la grilla se conectan con sus correspondientes en el extremo opuesto, formando una estructura toroidal. El computador denominado *Finite Element Machine* (FEM), ha sido desarrolado en la NASA, y su arquitectura responde a una disposición en grilla con 8 vecinos por nodo (fig. 7.1c). Otros diseños habituales, que se utilizaron en las instancias de pruebas, son las arquitecturas en cubo (fig. 7.2a) y en hibercubo (fig. 7.2b).

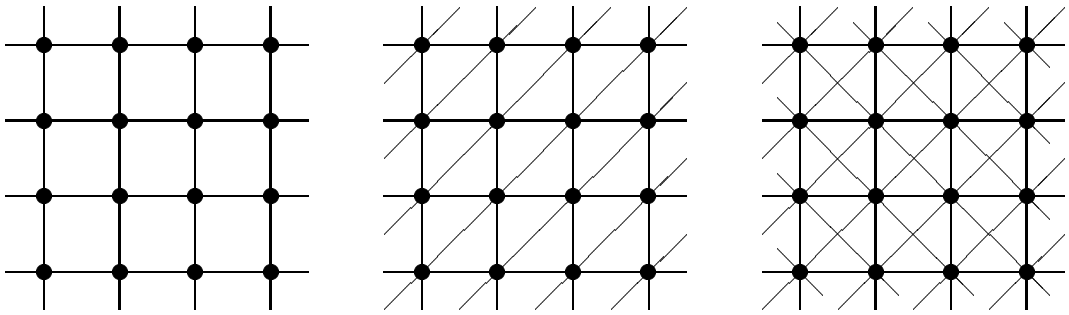


Figura 7.1: Arquitecturas de 4×4 en grilla con (a) 4, (b) 6 y (c) 8 vecinos por nodo.

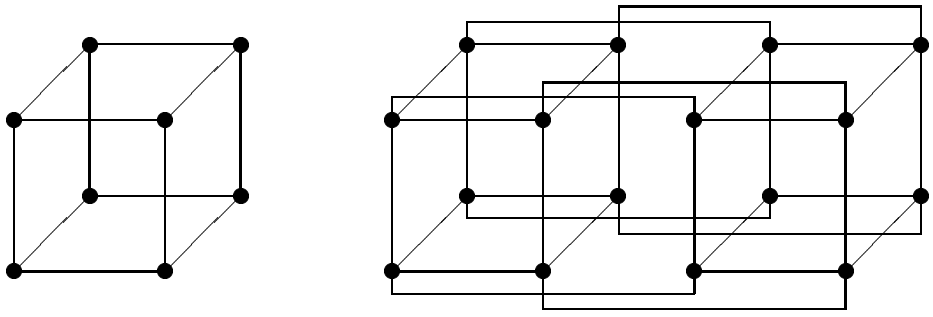


Figura 7.2: Arquitecturas en (a) cubo y (b) hipercubo.

Grupo de problemas STR. Estas instancias fueron propuestas por Bokhari (cfr. [Bok/81], donde son denominadas *problemas estructurados*). La figura 7.3a muestra el grafo de tareas de estos problemas, que tienen como grafo de procesadores arquitecturas en grilla con 8 vecinos

por nodo.

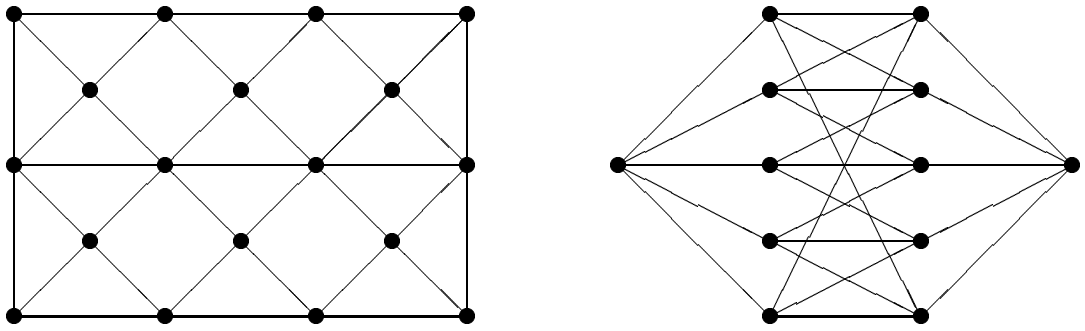


Figura 7.3: Grafos de tareas de los problemas (a) STR y (b) DF.

Grupo de problemas DC. En estas instancias, el grafo de tareas está formado por un árbol binario completo, y se utilizaron como grafos de procesadores a las distintas arquitecturas en grilla. Estas instancias son adaptaciones de problemas similares de la biblioteca de problemas ANDES, para el problema de planificación de tareas con restricciones de precedencia.

Grupo de problemas DF. Al igual que el grupo anterior, estas instancias son adaptaciones de instancias similares de la biblioteca ANDES. La figura 7.3b muestra el grafo de tareas de estas instancias. Cada nodo de las filas centrales se conecta con los tres nodos más cercanos de la fila opuesta, a excepción de los extremos, que también se conectan con los extremos opuestos. Nuevamente, se utilizó este grafo de tareas sobre arquitecturas en grilla.

Grupo de problemas WARS. Este grupo está formado por problemas de 8 y 16 nodos, sobre arquitecturas en cubo e hipercubo. Fueron propuestos en [Lee/87] como ejemplos que "aparecen en situaciones reales".

Grupo de problemas GAUSS. Las instancias de este grupo modelan la situación que se presenta cuando se resuelven sistemas de ecuaciones lineales utilizando el método de Gauss en forma paralela. El grafo de tareas está dividido en varios niveles, según puede verse en la figura 7.4. Se utilizaron arquitecturas en grilla como grafos de procesadores.

Grupo de problemas PRB. Este grupo está formado por instancias pequeñas (de hasta 12 nodos). Algunas de estas instancias se generaron al azar, mientras que otras se construyeron "a mano", con características especiales (árboles, grafos hamiltonianos, etc.). Se utilizaron, junto con otras instancias, durante la etapa de pruebas.

Grupo de problemas MEMSY. Este grupo fue presentado en [Mon/95]. Sus grafos de tareas son grillas rectangulares de 4 vecinos por nodo, y en todos los casos la arquitectura está formada por un grupo de 6 procesadores piramidales, organizados en dos niveles.

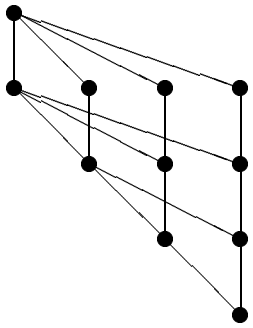


Figura 7.4: Grafo de tareas de los problemas GAUSS.

7.1.1 Grupo de problemas test

Seleccionamos un conjunto de 20 problemas test para realizar las pruebas computacionales de parámetros y estrategias. Estos problemas se eligieron de acuerdo con los siguientes criterios (cfr. [Jun/92]): En primer lugar, las instancias no deben ser demasiado pequeñas, dado que su resolución puede ser muy sencilla, conspirando contra los resultados de las pruebas. En segundo término, los problemas deben tener diferentes tamaños y estructura. En tercer lugar, estos problemas deben poder resolverse en tiempo razonable, dado que se realizará un gran número de pruebas. Por último, se observó que la heurística primal encuentra el óptimo en muchos casos, disminuyendo sensiblemente el tiempo de resolución. Por estos motivos, se eligieron problemas que pudieran ser resueltos sin el uso de esta heurística, para testear solamente el proceso *branch&cut* sin "interferencias" externas.

No se puede garantizar que los resultados obtenidos sean válidos para otros grupos de problemas con distinto tamaño y estructura, pero las conclusiones a las que se arribe proporcionarán una buena base para conocer el comportamiento del algoritmo. La tabla 7.1 describe el conjunto de 20 instancias de prueba. La columnas marcadas "Tareas" indican la cantidad de ejes del grafo de tareas y su densidad, mientras que las últimas dos columnas hacen lo propio con el grafo de procesadores.

7.2 Experimentos con las desigualdades válidas

Esta sección resume el primer grupo de pruebas, que se orientó a determinar la fuerza empírica de las desigualdades válidas encontradas. La fuerza "teórica" está dada por la dimensión de la cara que definen, pero puede suceder (y efectivamente sucede) que algunas desigualdades que no son a priori demasiado fuertes tengan en la práctica una contribución importante.

Problema	Nodos	Tareas		Procesadores	
DC2	15	14	13.33 %	30	28.57 %
DF1	12	25	37.88 %	18	27.27 %
PRUEBA3	5	5	50.00 %	6	60.00 %
PRUEBA4	5	6	60.00 %	6	60.00 %
PRUEBA5	8	10	35.71 %	10	35.71 %
PRUEBA6	8	13	46.43 %	10	35.71 %
PRUEBA7	10	16	35.56 %	14	31.11 %
PRUEBA7B	8	13	46.43 %	12	42.86 %
PRUEBA8	11	23	41.82 %	15	27.27 %
PRUEBA8B	8	10	35.71 %	10	35.71 %
PRUEBA9	11	17	30.91 %	15	27.27 %
PRUEBA9B	8	13	46.43 %	10	35.71 %
STR1	9	36	38.89 %	14	100.00 %
STR2	6	15	53.33 %	8	100.00 %
WARS1	8	10	35.71 %	12	42.86 %
WARS2	8	9	32.14 %	12	42.86 %
WARS3	8	13	46.43 %	12	42.86 %
WARS4	8	12	42.86 %	12	42.86 %
WARS5	8	13	46.43 %	14	50.00 %
GAUSS1	12	16	24.24 %	24	36.36 %

Tabla 7.1: Características de los problemas test.

7.2.1 Pruebas sobre la relajación lineal

Una forma de medir la fuerza de una familia de desigualdades consiste en agregar todas las desigualdades de la familia al modelo y resolver su relajación lineal. Como se trata de un problema de maximización, la relajación lineal tendrá un valor mayor o igual que el óptimo entero del problema, y las desigualdades válidas pueden hacer que el óptimo de la relajación disminuya. Cuanto mayor sea esta disminución, más "fuertes" serán las desigualdades. Debe notarse que estas afirmaciones son estrictamente empíricas, y que pueden no corresponderse con la fuerza teórica de las desigualdades, dada por la dimensión de la cara del poliedro que definen.

Se realizó este experimento con todos los problemas test[1], y se calculó para cada uno el cociente

$$gap = \frac{z_{cuts} - z_{LP}}{z_{IP} - z_{LP}},$$

donde z_{LP} y z_{cuts} representan los óptimos de la relajación lineal, con y sin la familia de desigualdades, y z_{IP} indica el óptimo entero. Este indicador se mueve entre 0 y 1, siendo 0 cuando $z_{cuts} = z_{LP}$ (las desigualdades no mejoran el óptimo de la relajación) y 1 cuando $z_{cuts} = z_{IP}$ (las desigualdades logran el valor del óptimo entero). Para cada familia, se indica

[1] Debe mencionarse que en las familias con una cantidad exponencial de desigualdades no fue posible realizar este experimento con las tres instancias más grandes, dado que se agotó la memoria virtual al intentar agregar todas las desigualdades.

en la tabla 7.2 el promedio de estos cocientes sobre todos los problemas, que proporciona una idea de la fuerza de la familia. Se muestra además la cantidad de desigualdades de cada familia (n indica la cantidad de nodos del problema, m la cantidad de ejes del grafo de tareas y $\bar{m}_H$ la cantidad de ejes del complemento del grafo de procesadores).

Familia	Cantidad	Factor	Observaciones
1	$m\bar{m}_H$	0.0000	Restricciones del primer modelo
2	$n2^n$	**0.9095**	Obtenidas por inspección
3	n	**0.7720**	Cortan a x^{inf}
4	nm	**0.0297**	Definen facetas
5	nm	0.0297	Definen facetas
6	nm	**0.7598**	—
7	$\leq m2^n$	0.0000	Busca conjuntos independientes
9	n	0.0000	—
10	$\leq n2^n$	**0.0297**	Generalización de (2), sin *lifting*
11	$m2^n$	**0.0340**	Definen facetas
12	$m2^n$	**0.0396**	Definen facetas

Tabla 7.2: Pruebas con las familias de desigualdades.

Estos resultados presentan muchas particularidades que pueden resaltarse. En primer lugar, observemos que el óptimo de la relajación lineal siempre estará dado por la cantidad m de ejes del grafo de tareas, dado que el "punto infausto" x^{inf} (que es fraccionario) cumple las restricciones del modelo y tiene función objetivo igual a m. Como la función objetivo es $\sum c_{ij} \leq m$, entonces el óptimo de la relajación tendrá este valor.

La tabla muestra que al agregar la primer familia de desigualdades no mejora el óptimo de la relajación lineal. Estas desigualdades son las restricciones de costo del primer modelo, que también admiten a x^{inf}, con lo cual al agregarlas no se corta a este punto, con función objetivo m. Podría pensarse que estas restricciones son implicadas por las restricciones de costo del segundo modelo, pero esto no sucede, dado que durante el algoritmo *branch&cut* se generan cortes a partir de estas desigualdades (cfr. siguiente subsección).

Por otra parte, se observó una notable disminución en el óptimo de la relajación cuando se agregaron todas las desigualdades de la segunda familia. Estas desigualdades se obtuvieron por inspección de las soluciones arrojadas por la relajación lineal en instancias pequeñas, y cortan bajo condiciones generales estas soluciones (cfr. sección 5.4). En un primer momento, la relajación lineal junto con esta familia de desigualdades arrojó siempre el valor óptimo (en muchos casos no se halló el óptimo entero, sino una solución fraccionaria con el mismo valor). Se hicieron pruebas adicionales que mostraron que esto no siempre sucede. Sin embargo, la performance empírica de estas desigualdades es notable, siendo que no parecen definir facetas del poliedro.

Debe mencionarse el importante resultado logrado por las desigualdades de la familia 3. Estas desigualdades tienen especial importancia puesto que cortan a x^{inf} en las arquitecturas habituales. Sin embargo, podrían subsistir otros puntos con igual función objetivo fuera del alcance de estas desigualdades, lo cual no sucede en general, dados los valores obtenidos. Es importante recordar que esta familia está compuesta por **sólo n desigualdades**. En el

problema PRB8, se pasa de 23 (óptimo de la relajación lineal) a 14 (óptimo de la relajación junto a esta familia). En los problemas STR1 y WARS4 se pasa de 35 a 14, y de 12 a 6 respectivamente, y se obtienen resultados similares, aunque no tan dramáticos, en los otros problemas test.

Por último, dentro del grupo de familias que lograron los mejores resultados, es preciso notar que la familia 6 obtuvo una disminución promedio del 17 % respecto del óptimo de la relajación lineal. Otras familias de desigualdades obtuvieron mejoras, aunque no fueron tan significativas como las logradas por las familias 2, 3 y 6. Entre estas familias que lograron pequeñas mejoras se cuentan todas las familias de facetas.

Además del análisis de la mejora de cada familia individual, surge una particularidad al comparar los resultados de las familias 2 y 10. Dada una tarea i, estas desigualdades surgen de considerar las tareas vecinas. La familia 2 considera a todos los vecinos y la familia 10 tiene en cuenta a *todos los subconjuntos de vecinos*, y constituyen una generalización de la primera. Ahora bien, para obtener desigualdades válidas para todo el poliedro, en el caso de la familia 2 se recurre a un procedimiento fuerte de *lifting* sobre varias variables, mientras que en la familia 10 se añade un término suficientemente grande, que se anula cuando es necesario (es decir, se trata de un *lifting* más débil). Como muestra la tabla, a pesar de que la familia 10 generaliza a la familia 2, el procedimiento de *lifting* fuerte realizado sobre esta última logra una mejor performance.

Para finalizar el análisis de estos resultados, debe mencionarse que existen familias que no mejoran el óptimo de la relajación lineal. Entre ellas se encuentra la familia 7, formada por un número exponencial de desigualdades. Como se mencionó en la sección 5.4, la cara definida por estas desigualdades tiene dimensión baja, dado que se cumplen por igualdad en condiciones muy restrictivas. Sin embargo, aunque estas desigualdades no mejoren el óptimo de la relajación lineal, pueden ser útiles como cortes, y la experiencia muestra que esto efectivamente sucede.

7.2.2 Mediciones de los cortes generados

El análisis realizado en la sección anterior es un análisis estático, dado que se verifica la situación que se produce al agregar toda una familia de desigualdades a la relajación lineal del modelo. Estas pruebas consideran a todas las desigualdades de una familia en bloque, como una forma de medir su fuerza por métodos empíricos. Sin embargo, durante la ejecución de un algoritmo *branch&cut*, las desigualdades se agregan a medida que se detectan, en un proceso dinámico que las genera.

Tiene particular interés conocer la cantidad de desigualdades de cada tipo que se producen durante la ejecución. Por ejemplo, una familia de desigualdades podría ser muy buena pero puede no ser generada durante el algoritmo porque los óptimos sucesivos la cumplen, o bien porque los procedimientos de separación no la detectan. En este sentido, se realizaron mediciones de la cantidad de desigualdades de cada tipo que se producen durante la ejecución de las instancias de problemas test, y los resultados se muestran en la tabla 7.3. Cada familia tiene asociado un procedimiento de separación, que puede hallar hasta una cierta cantidad

Problema	Subpr	1	2	4	5	6	7	8	10	11	12	Regen
DC2	1001	0.00	17.00	0.60	0.00	19.92	17.78	0.51	31.55	8.33	11.23	75.58
DF1	1229	0.00	4.27	0.50	0.10	45.83	9.52	0.67	44.15	17.16	9.14	88.16
PRB3	1	0.00	0.00	0.00	0.00	20.00	0.00	0.00	60.00	0.00	0.00	0.00
PRB4	15	0.00	6.66	0.44	1.11	34.66	5.55	0.00	32.00	2.66	3.33	13.92
PRB5	37	0.00	13.17	1.55	0.00	44.93	24.86	0.81	40.54	8.78	7.56	39.50
PRB6	5	0.00	12.50	1.53	0.00	60.00	20.00	1.53	55.00	10.00	15.38	17.92
PRB7	201	0.00	7.66	0.92	0.12	49.20	18.09	0.65	46.96	14.22	9.48	78.96
PRB7B	63	0.00	5.95	1.14	0.24	42.65	14.28	0.00	46.23	7.14	4.51	55.15
PRB8	977	0.16	4.45	0.62	2.06	44.35	13.07	0.42	45.14	13.92	7.04	77.86
PRB8B	31	0.32	11.69	1.25	2.25	28.22	16.45	1.29	41.12	9.27	8.38	28.03
PRB9	243	0.00	7.18	0.91	0.07	48.29	16.72	0.48	46.72	12.94	9.41	80.53
PRB9B	451	0.23	3.99	1.03	0.34	48.17	11.90	0.40	47.22	8.89	6.39	74.84
STR1	593	0.03	0.31	0.45	1.00	49.91	6.68	0.03	49.10	15.85	4.72	57.10
STR2	17	0.00	4.90	1.04	2.74	47.05	12.54	0.00	46.07	6.86	4.70	19.44
WARS1	17	0.00	12.50	1.54	0.00	38.97	26.47	0.00	43.38	12.50	7.05	32.93
WARS2	61	1.45	7.58	1.34	0.18	26.22	24.77	0.18	30.32	5.94	4.00	50.18
WARS3	309	1.29	4.73	1.35	0.00	49.83	20.36	0.54	44.25	9.50	6.22	71.69
WARS4	57	0.00	3.72	0.45	0.73	14.03	12.42	0.00	44.73	2.63	2.63	25.86
WARS5	4249	0.52	4.38	0.64	2.77	32.57	6.35	0.05	47.72	8.49	4.88	70.42
GAUSS1	1523	5.60	13.20	1.57	0.04	28.77	25.81	1.21	32.26	12.05	10.90	66.80
Promedio		0.48	7.29	0.94	0.69	38.68	15.18	0.44	43.72	9.36	6.85	51.24

Tabla 7.3: Porcentaje de éxito de los procedimientos de separación.

de cortes por subproblema (por ejemplo, muchos procedimientos buscan una desigualdad por eje, con lo cual puede hallar un máximo de m cortes por ejecución). Para cada ejecución de los procedimientos de separación, se calculó su porcentaje de efectividad (cortes generados vs. máxima cantidad de cortes posible), y se muestra en la tabla el promedio sobre todos los subproblemas. Se indica además la cantidad de nodos del árbol de enumeración (primer columna), y el porcentaje de cortes regenerados desde el *pool* de desigualdades sobre el total de cortes (última columna). Debe notarse que esta tabla no incluye las familias 3 y 9 (formadas por un número lineal de desigualdades), puesto que no tienen procedimientos de separación asociados, sino que se ingresan al *pool* de desigualdades al comienzo del proceso.

Una primera observación que debe realizarse concierne a la cantidad de cortes producidos por el proceso de *regeneración*. Cuando se presentó este procedimiento, se mencionó el número de cortes que produce como una de las razones para su inclusión, y esta tabla confirma estas afirmaciones. Puede verse que el procedimiento de regeneración es responsable del 51.24% de los cortes totales. Es decir, la cantidad de cortes producidos por este método supera levemente la cantidad total de desigualdades generadas por los procedimientos de separación, lo cual justifica su uso. En efecto, las desigualdades que se separan en forma heurística pueden ser reutilizadas en otro momento, aunque los procedimientos respectivos no las encuentren.

Por otra parte, puede verse que la separación sobre las familias exponenciales de desigualdades tiene un porcentaje alto de efectividad, destacándose la familia 10 con un 43.57% de éxito (esto significa que, en promedio, se logran $\approx 0.4m$ cortes de esta familia por subproblema). Por su parte, las familias 11 y 12, que constan de un número exponencial de facetas, muestran también un alto porcentaje de efectividad en la separación, aportando en conjunto el 16.09% de los cortes, que cobran importancia al tratarse de las desigualdades más fuertes

posibles (desde el punto de vista teórico). Deben mencionarse también las familias 2 y 7, cuyos procedimientos de separación son heurísticos y logran detectar un gran número de cortes. Estas familias no definen facetas en general, y no parecen constituir cortes demasiado fuertes (de acuerdo con las propiedades de las caras que definen). Sin embargo, los resultados muestran que su aporte es importante.

La familia 8 es la única exponencial que no aporta muchos cortes. Tiene muchos casos particulares en los cuales las desigualdades son redundantes o pueden ser inmanejables, pero se fijó un subgrupo de estas desigualdades (aquellas formadas tomando 5 conjuntos independientes) que es exponencial y no es redundante (cfr. sección 6.1.8). Sin embargo, el procedimiento heurístico de separación no encuentra una gran cantidad de cortes, como lo muestra la tabla, además de tener un costo computacional que puede ser elevado.

Debe notarse también la gran performance de la familia 6, formada por un número polinomial de desigualdades. Sin embargo, debe tenerse en cuenta que el procedimiento de separación de esta familia sobreestima los coeficientes de *lifting* por razones de eficiencia, haciendo que puedan agregarse desigualdades que no corten a la solución actual (se calculan los coeficientes exactos para las desigualdades que sean cortes).

Finalmente, es preciso notar la contribución de las desigualdades de la primer familia, formada por las restricciones de costo del primer modelo. Al constituir cortes, se prueba que estas restricciones no se deducen de las restricciones del segundo modelo, que siempre están presentes en las formulaciones de los subproblemas. Sin embargo, su aporte como desigualdades válidas no es muy grande, en cuanto a la cantidad de cortes que generan.

7.2.3 Performance de las familias de desigualdades

En las pruebas que se realizan en esta sección, se busca conocer el aporte de cada familia durante la ejecución del algoritmo *branch&cut*, junto con los procedimientos de separación. Se realizaron ejecuciones de las instancias de problemas test utilizando distintas combinaciones de familias, para comprobar la variación en el tiempo de ejecución que estas modificaciones producen. Para estas pruebas, se dividen las familias en dos clases según la cantidad de desigualdades que contengan. Consideremos entonces el conjunto FP $= \{1, 3, 4, 5, 6, 9\}$ de familias con un número polinomial de elementos y el conjunto FE $= \{2, 7, 8, 10, 11, 12\}$ de familias exponenciales. Uno de los objetivos de las pruebas fue medir la influencia de cada grupo de familias sobre el tiempo total de ejecución.

Las instancias del grupo de problemas test son diferentes entre sí, y su resolución arroja tiempos dispares. En [Jun/92] se utiliza la suma de los tiempos como una medida para estimar el tiempo para un cierto valor del parámetro. Sin embargo, con tiempos muy distintos entre sí no se obtiene una buena medida para comparar, dado que los tiempos más pequeños estarán dominados por los tiempos mayores, y su sumatoria solamente reflejará lo que sucede con los problemas que necesitan mayor tiempo. Para que esto no suceda, se "normalizan" los tiempos de resolución de cada instancia, dividiéndolos por el tiempo utilizado tomando las desigualdades del conjunto FE+3 (fila 9). Con esto, se tiene una medida más adecuada para comparar los tiempos de resolución (se realiza el mismo proceso con los otros indicadores).

La tabla 7.4 resume los resultados de estas pruebas. Como se indicó, se normalizan los resultados de cada instancia con relación a la fila 9 de la tabla, y se muestran los promedios de estas normalizaciones. La última columna indica las familias que se incluyeron en cada caso. Por ejemplo, "FP + 2" significa que se consideraron todas las familias de FP junto con la familia 2, y "FP + FE - 11" indica que se incluyeron todas, excepto la familia 11.

Set.	Tiempo	#Subpr.	#LPs	Altura	Familias
1	3.7652	3.3531	3.1874	1.2893	FP
2	3.0368	2.7217	2.5901	1.2090	FP+4
3	3.0474	2.6040	2.4285	1.1728	FP+10
4	4.5649	3.9438	3.7685	1.3051	FP+11
5	4.0376	3.5591	3.3574	1.3072	FP+15
6	3.5778	3.1345	2.9659	1.2835	FP+16
7	3.3707	2.8537	2.6342	1.2790	FP+17
8	**1.1040**	1.0414	1.0385	1.0070	FE
9	**1.0000**	1.0000	1.0000	1.0000	FE+3
10	**1.3736**	1.2843	1.2921	1.0586	FE+7
11	**0.9983**	1.0558	1.0575	1.0030	FE+8
12	2.2087	1.7452	1.5986	1.1472	FE+9
13	2.9770	2.2937	2.1854	1.1979	FP+FE-4
14	2.6197	2.1250	1.9608	1.1708	FP+FE-10
15	2.6750	1.9783	1.8708	1.1834	FP+FE-11
16	2.0008	1.6872	1.5820	1.1463	FP+FE-15
17	2.6003	1.9035	1.8051	1.1456	FP+FE-16
18	2.8093	2.2730	2.1701	1.1778	FP+FE-16
19	2.5976	1.9287	1.8162	1.1631	FP+FE

Tabla 7.4: Performance utilizando distintas familias de cortes.

La primera observación que surge de esta tabla se verifica en relación a los menores tiempos, que se obtuvieron con las ejecuciones 8 a 11. Puede verse que *se obtiene uno de los mejores tiempos si se utilizan solamente las familias exponenciales de desigualdades*, y que este tiempo se mejora levemente al agregar las familias 3 y 8. Excepto el grupo 12 (FE+9), al considerar casi exclusivamente las familias exponenciales se logran los mejores resultados.

Pueden agruparse las filas de la tabla en tres grandes grupos: (a) las filas 1 a 7, que consideran FP junto con alguna familia de FE adicional; (b) las filas 8 a 12, que consideran FE con alguna familia de FP adicional; y (c) las filas 13 a 19, que consideran todas las familias, sacando una en cada paso. Resulta notable la diferencia existente entre estos grupos, y puede verse que dentro de cada grupo los resultados se mueven dentro de ciertas franjas de valores:

1. El grupo (b) considera solamente las familias exponenciales, agregando una familia polinomial por vez (filas 8 a 12). Con excepción de la fila 12, los resultados de estas ejecuciones se encuentran entre 0.9 y 1.4, con lo cual constituyen las mejores performances de estas pruebas.

2. En segundo término, el grupo (c) considera todas las familias, quitando una familia de FE cada vez (filas 13 a 19). Estas filas arrojan resultados que se mueven entre 2.0 y 2.9.

3. Por último, si (c) solamente utilizamos las familias polinomiales (agregando quizás una familia de FE), se obtienen tiempos en el rango [3.0, 4.5].

Estos valores permiten formar una idea aproximada del comportamiento del algoritmo con diferentes combinaciones de familias. Puede verse que los mejores tiempos se obtienen con las familias exponenciales, y que si se agregan las familias polinomiales, el tiempo aumenta. Por su parte, si sólo tenemos en cuenta las familias de FP, los tiempos son mucho mayores.

Debe notarse un hecho inesperado en estos resultados. Al considerar una mayor cantidad de familias, cabría esperar que el tamaño de los árboles disminuyese, aunque el tiempo fuera mayor. Sin embargo, se observan tamaños mayores con FP+FE que con FE, que están acompañados también de una mayor cantidad de problemas lineales resueltos y una mayor altura del árbol de enumeración. Estos resultados no se esperaban a priori, pero no pueden pasarse por alto. Se realizaron nuevas pruebas que confirmaron estos datos.

7.2.4 Cortes por subproblema

Entre los indicadores de la performance del algoritmo, la cantidad de cortes por subproblema es de interés, y se realizaron mediciones para conocer su comportamiento. En la primera fase de la separación, se regeneran cortes del *pool*, y en la segunda se aplican los procedimientos de separación propiamente dichos para cada familia de desigualdades (cfr. sección 6.1.13). Se midieron los resultados de cada fase por separado, para conocer el aporte de cada una al total de cortes generados.

Procedimientos de separación. En primer lugar, consideremos la segunda fase de la separación. En cada subproblema se genera una cierta cantidad de cortes, y el gráfico 7.5 muestra la distribución de estas cantidades. Es decir, para cada valor t se grafica el número de nodos en los cuales se generaron t cortes con los procedimientos de separación específicos. El gráfico muestra estos valores para cuatro problemas test, de distinta estructura y tamaño similar. Puede verificarse en este gráfico que se generan cortes en prácticamente todos los nodos, y que la cantidad de cortes se sitúa dentro de un rango definido (aproximadamente entre 25 y 45 desigualdades por nodo, para estos problemas).

Regeneración de cortes. En la primera etapa de la separación, se regeneran cortes del *pool*, verificando si alguna de las desigualdades halladas en nodos anteriores es violada en el óptimo actual. El gráfico 7.6 muestra las distribuciones de la cantidad de cortes regenerados para los mismos problemas que el gráfico anterior, pero escalado en 1:10 unidades para el eje de las abscisas (es decir, el punto (x, y) en el gráfico indica que en y nodos se regeneraron entre $10x$ y $10x + 9$ cortes, inclusive). En primer lugar, se observan rangos más amplios, pero que también se concentran dentro de una zona determinada.

Estas pruebas parecen indicar que el número de cortes generados por subproblema se mantiene aproximadamente dentro de un cierto rango. Es decir, el proceso de separación "tiende" a encontrar cantidades similares de cortes en los nodos del árbol.

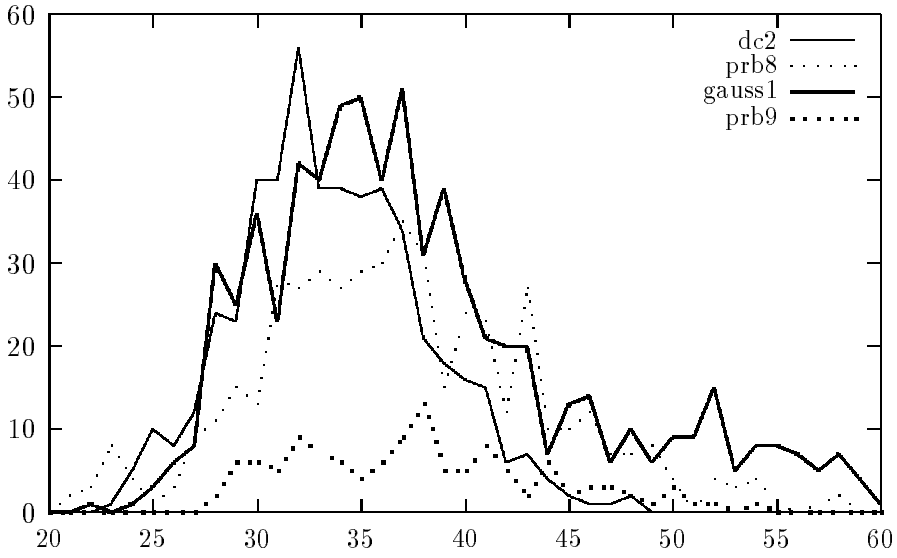


Figura 7.5: Distribución de la cantidad de cortes por subproblema.

7.3 Ajuste de los parámetros

En esta sección se describe el segundo grupo de pruebas computacionales, que conforman el proceso de ajuste de parámetros del algoritmo. Los algoritmos *branch&cut* tienen una gran cantidad de parámetros asociados, y el tiempo de ejecución y el tamaño del árbol pueden variar en gran medida de acuerdo a la combinación de parámetros utilizada. En la mayoría de los casos, el comportamiento del algoritmo al variar los parámetros responde a las expectativas y puede explicarse razonablemente, a pesar de la enorme cantidad de factores involucrados.

7.3.1 Iteraciones por Subproblema

Cada nodo del árbol de enumeración representa un subproblema del problema inicial. En cada uno de ellos, se resuelve la relajación lineal del subproblema y, si su óptimo no es entero, se generan planos de corte que se agregan a su formulación. Este proceso (resolución del subproblema y generación de cortes) se denomina una *iteración*. Uno de los parámetros generales del proceso es la cantidad de iteraciones que se realizan por cada subproblema, y las pruebas descriptas en esta sección se orientaron a conocer el comportamiento del algoritmo con distintos valores de este parámetro. La tabla 7.5 resume los resultados obtenidos para los 20 problemas test, normalizados nuevamente a la primera fila, para distintas cantidades de iteraciones. Para cada indicador, se presenta el promedio de los valores normalizados de los problemas.

A medida que se incrementa la cantidad de iteraciones por nodo, el tiempo de procesamiento total (segunda columna) aumenta progresivamente, hasta llegar a ser casi 6 veces superior cuando se realizan 10 iteraciones por nodo. Este incremento del tiempo está originado por el aumento en la cantidad de veces que se utilizan los procedimientos de separación,

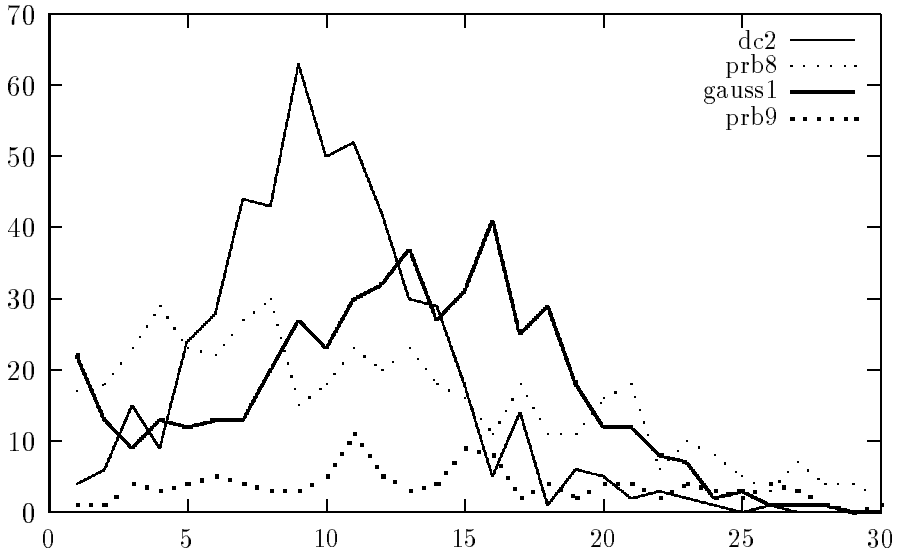


Figura 7.6: Distribución de la regeneración de cortes por subproblema.

Iters.	Tiempo CPU	Tiempo Sep.	Nodos B& C	Altura árbol	# Cortes
1	1.0000	1.0000	1.0000	1.0000	1.0000
2	1.4930	2.4805	0.9405	0.9819	1.6001
3	1.9228	3.6601	0.8726	0.9433	1.9557
4	2.7159	6.8705	0.8941	0.9305	2.5811
5	3.2994	9.3166	0.8753	0.9107	2.8340
6	3.3260	9.0546	0.7792	0.9030	2.7846
7	4.3288	13.7945	0.8155	0.8708	3.2119
8	5.2044	16.8230	0.8530	0.8619	3.6431
9	4.7363	16.1141	0.7242	0.8655	3.3613
10	5.9206	19.9608	0.7912	0.8533	3.8098

Tabla 7.5: Resumen por cantidad de iteraciones (base: 1 iteración)

cuyo tiempo aumenta considerablemente (tercera columna). Por su parte, este incremento en la cantidad de iteraciones por nodo permite que se agregue una mayor cantidad de cortes (última columna), con lo cual el tamaño y la altura del árbol de enumeración disminuyen (columnas 4 y 5).

En resumen, a medida que se consideran más iteraciones por nodo del árbol de enumeración, se disminuye su tamaño al agregarse una mayor cantidad de cortes, pero el tiempo de ejecución es progresivamente mayor. Por lo tanto, se fija este parámetro en 1 iteración por nodo.

7.3.2 Estrategias de *branching* y selección de nodos

Luego de realizar un cierto número de iteraciones en un nodo, si el subproblema asociado es factible con óptimo no entero, se generan dos nodos hijos mediante el procedimiento de *branching*. Se consideraron tres estrategias posibles para realizar este proceso. Si se realiza *branching* por variables, puede (a) tomarse la variable fraccionaria con valor más cercano a 0.5, o (b) elegir la variable con valor más cercano a 0.5 entre las que tengan mayor coeficiente en la función objetivo. Estos criterios se denominan *Close Half* (CH) y *Close Half Expensive* (CH Exp.) respectivamente. Por otra parte, (c) puede utilizarse el *branching* refinado, descripto en la sección 6.3.

Una vez generados los hijos del nodo actual, se debe seleccionar un nuevo nodo de la lista de subproblemas activos para continuar el proceso. Nuevamente, existen distintos criterios para elegir este nodo: puede (a) seleccionarse aquel cuya relajación tenga mejor función objetivo, (b) recorrer el árbol en BFS, o (c) hacerlo en DFS. La estrategia de *branching* y el modo de selección del siguiente nodo constituyen dos parámetros fundamentales del algoritmo, y se realizaron pruebas para medir la performance del algoritmo utilizando todas sus combinaciones. Estas pruebas tienen especial interés, puesto que permitirán comparar empíricamente el criterio de *branching* habitual (por variables) con el procedimiento refinado.

La tabla 7.6 resume los resultados de las pruebas. Cada fila muestra el promedio de las mediciones para los problemas test (normalizados a la primera fila) bajo una combinación de estos parámetros. Las dos primeras columnas indican el criterio de *branching*, y el criterio de selección de nodos, respectivamente.

Branch	Sel. Nodo	CPU (sg.)	LP (sg.)	Sep. (sg.)	# Subpr.	Altura
CH	Best	1.0000	1.0000	1.0000	1.0000	1.0000
CH	BFS	1.6420	1.6211	2.2653	1.5754	0.9966
CH	DFS	1.1585	1.1934	1.3296	1.0385	1.0107
CH Exp.	Best	3.6533	2.8875	12.7753	2.8444	1.2338
CH Exp.	BFS	40.3200	15.2558	294.9095	16.5371	1.0792
CH Exp.	DFS	42.4389	28.1428	190.2787	19.5950	1.6633
Ref.	Best	22.3751	14.7448	80.8148	11.4913	1.5029
Ref.	BFS	25.6243	16.1167	101.3850	13.3597	1.4766
Ref.	DFS	4.2809	4.2708	6.6586	3.5643	1.9354

Tabla 7.6: Resultados de distintas estrategias.

La tabla muestra resultados dispares entre las estrategias. En primer lugar, debe notarse que la estrategia de *branching* refinado (últimas tres filas) no produce mejores resultados que el *branching* normal por variables. En particular, el criterio de seleccionar la variable con valor más cercano a 0.5 (CH) arroja los mejores tiempos, independientemente de la estrategia de selección de nodos.

Se decidió utilizar la combinación que produce mejores resultados. De acuerdo con estas pruebas, se utiliza *branching* por variables, tomando como variable de *branching* aquella con valor más cercano a 0.5, junto con la estrategia *Best First* para la selección de nodos del árbol

de enumeración.

7.3.3 Ajuste del parámetro *skip factor*

Una de las decisiones más importantes en cuanto al ajuste de los parámetros del algoritmo está ligada con el manejo de los procedimientos de separación. Estos procedimientos permiten identificar desigualdades violadas, que se agregan a la formulación de los nodos, pero tienen un costo computacional que puede ser elevado. Esto se agrava en el caso de procedimientos de separación heurísticos, que pueden inclusive no encontrar ninguna desigualdad. Puede suceder que sea beneficioso no utilizar siempre estos procedimientos, al costo de generar una menor cantidad de desigualdades violadas. Una estrategia muy utilizada consiste en generar cortes cada una cierta cantidad de nodos del árbol, y en esta sección se describen las pruebas realizadas en este sentido.

Skip factor por nodo. El parámetro *skip factor* indica cada cuántos nodos del árbol de enumeración se generan cortes, llamando a los procedimientos de separación. Si este parámetro se fija en 1, entonces se generan cortes luego de cada problema lineal, pero los procedimientos de separación tienen un costo computacional que puede hacer inconveniente esta situación. Si en lugar de esto se generan cortes, por ejemplo, cada 2 nodos, la cantidad de cortes agregados será menor, pero puede suceder que el tiempo total de ejecución disminuya, al realizar menos llamadas a los procedimientos de separación. La tabla 7.7 resume los resultados obtenidos para el grupo de problemas test, "normalizados" a los valores obtenidos con *skip factor* igual a 1.

Skip Factor	CPU (sg.)	Sep. (sg.)	Nodos B&C	LPs	Altura	Cortes
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	1.00160	0.94516	1.63791	1.19225	1.10696	0.89030
3	**0.84723**	0.44800	1.72062	1.08267	1.09644	0.67312
4	**0.84166**	0.44206	1.89542	1.09992	1.16200	0.56670
5	**0.81357**	0.34449	2.00892	1.10762	1.15187	0.48364
6	**0.84669**	0.31691	2.16938	1.22655	1.17216	0.44436
7	0.96609	0.31056	2.47536	1.31049	1.24694	0.46745
8	0.95845	0.29822	2.70064	1.27914	1.23681	0.43860
9	0.93270	0.26922	2.50988	1.30751	1.23616	0.39214
10	1.04692	0.25791	2.86984	1.38444	1.27157	0.41032
11	1.72371	0.25563	3.38463	2.24625	1.22118	0.50342
12	1.20960	0.12342	2.25091	1.43445	1.13725	0.34838
13	1.68106	0.22766	3.31786	2.17808	1.21568	0.42663
14	1.66325	0.17146	3.58964	2.28192	1.25793	0.44306
15	1.61292	0.09524	3.45954	2.14311	1.23993	0.39999

Tabla 7.7: Resultados al variar el parámetro *skip factor*.

Los resultados muestran que efectivamente el tiempo de separación disminuye a medida que se aumenta el parámetro *skip factor*, dado que los procedimientos de separación se ejecutan con menor frecuencia. Otra consecuencia directa de este hecho es la menor cantidad

de cortes que se generan (última columna), que llega a ser un 60% menor que los valores iniciales. Además, al generar menos cortes, el tamaño del árbol aumenta, tal como reflejan las columnas correspondientes en esta tabla.

Ahora bien, al generar cortes con menos frecuencia se ahorra el tiempo que tomarían los algoritmos de separación, y esto puede mejorar el tiempo total de resolución, aunque el árbol de enumeración tenga más nodos. Como muestra la segunda columna de la tabla, el tiempo total de ejecución realmente disminuye para valores de *skip factor* entre 3 y 9, logrando disminuciones entre 15 % y 20 % cuando se toma este parámetro entre 3 y 6. Por lo tanto, el tiempo que se ahorra al generar cortes con menor frecuencia permite resolver los problemas en tiempo menor, aunque el tamaño del árbol sea mayor.

Skip factor por nivel. Una segunda posibilidad con relación al manejo del parámetro *skip factor* consiste en aplicar cortes salteando niveles del árbol. En las pruebas anteriores, el parámetro *skip factor* indicaba cada cuántos nodos se aplicaban los procedimientos de separación, mientras que ahora se saltean niveles completos del árbol. Si se toma este parámetro con valor s, entonces se aplican cortes en todos los nodos de los niveles $1 + ks$ ($k \geq 0$). La tabla 7.8 resume los resultados obtenidos, normalizados a la primera fila.

SF	Tiempo	Tiempo LP	Tiempo Sep.	# Subpr.	Altura
1	1.0000	1.0000	1.0000	1.0000	1.0000
2	0.9757	1.0584	0.7856	1.3269	1.0171
3	0.9396	1.1295	0.4217	1.4370	1.0162
4	0.8267	1.0487	0.3044	1.4156	1.0190
5	1.2003	1.5589	0.5159	2.2434	1.0652
6	0.9704	1.2526	0.2819	1.7663	1.0743
7	1.0777	1.3792	0.3136	2.0472	1.0768

Tabla 7.8: Resultados con *skip factor* por nivel.

Nuevamente, se obtienen mejores tiempos cuando no se aplican cortes en todos los nodos. Como puede verse, la cantidad de subproblemas y la altura del árbol aumenta progresivamente a medida que se disminuye la frecuecia de la generación de cortes. Sin embargo, el tiempo de separación también disminuye, y este ahorro de tiempo permite que la ejecución sea más rápida, aunque recorra árboles más grandes. En este caso, las mejoras más importantes se observan al saltear 3 y 4 niveles.

En conclusión, las mediciones realizadas al variar este parámetro muestran que pueden obtenerse tiempos beneficiosos al reducir la frecuencia de la generación de cortes. Las pruebas muestran que esto aumenta el tamaño del árbol, pero el menor tiempo dedicado a la separación permite compensar esta situación y obtener tiempos menores en algunos casos. Para la implementación definitiva, se fijó este parámetro en $sf = 5$, según la primera estrategia (es decir, se llama a los procedimientos de separación cada 5 nodos).

7.3.4 Heurística de mejoramiento primal

En cada nodo del árbol de enumeración se ejecuta una heurística primal, a partir de una solución entera construida por redondeo de la solución del subproblema. Se realizaron mediciones para verificar la efectividad de esta heurística, y se comprobó que en ningún momento este mecanismo aportó mejoras a la cota inferior del algoritmo. Si la heurística primal está desactivada, entonces este procedimiento cobra importancia, pero su aporte se anula al utilizar una heurística inicial. Se intentó utilizar CLO como heurística de mejoramiento primal, pero no se obtuvieron mejores resultados, por lo que se decidió desactivar este mecanismo para las pruebas finales.

7.3.5 Estrategias de regeneración de cortes

Cada vez que se genera un corte, la desigualdad correspondiente se guarda en el *pool* de cortes, de modo tal que éste contiene todas las desigualdades que se hallaron a lo largo del algoritmo. Esta información se utiliza en el proceso de separación, durante el cual puede revisarse este *pool* buscando desigualdades violadas en el óptimo actual. El procedimiento de separación tiene dos fases: (a) regeneración de cortes del *pool* y (b) aplicación de los procedimientos de separación de cada familia.

Set	Tiempo CPU	# Subpr.	# LPs	Altura	Cortes
1	1.0000	1.0000	1.0000	1.0000	1.0000
2	3.9923	4.6754	5.9099	1.4143	2.7488
3	1.2062	2.3052	2.3720	1.1774	0.4715
4	1.1045	1.8560	1.6804	1.1137	0.6492

Tabla 7.9: Comparación entre distintas estrategias.

Existen distintas estrategias sobre el manejo de estas dos fases (cfr. [Jun/98]), y las pruebas que se reseñan en esta sección se orientaron a determinar el impacto en el tiempo de ejecución de cada una de ellas. La tabla 7.9 muestra los promedios de los resultados para el conjunto de problemas test, normalizados a la primera fila. Cada fila resume los resultados con una estrategia distinta:

1. Estrategia normal. Se ejecutan las dos fases de la separación.
2. No se regeneran cortes del *pool* (la separación consta solamente de la segunda fase).
3. Si la primera fase puede regenerar cortes, entonces no se ejecuta la segunda fase (es decir, si se regeneran cortes del *pool*, no se aplican los procedimientos específicos de separación de cada familia). Esto garantiza que no haya entradas duplicadas en el *pool* de cortes.
4. Se asigna una probabilidad a cada familia, en base a la frecuencia de los cortes (tabla 7.3), y se ejecuta cada procedimiento de separación con la probabilidad asignada. Si la probabilidad del procedimiento t es p_t, se genera un número al azar $x \in [0,1]$ y se

separa la familia t si $x \leq p_t$. De esta forma, se espera que las familias más efectivas se separen con mayor frecuencia.

Como puede verse en la tabla, el proceso de regeneración de cortes del *pool* es muy importante, dado que se obtienen tiempos (y árboles) mucho mayores cuando este mecanismo se desactiva (fila 2). La tercera estrategia (no realizar la fase 2 cuando se regenera al menos un corte) produce menos cortes y, consecuentemente, árboles más grandes (2.3 veces mayores, en promedio). Sin embargo, el tiempo que se ahorra al no ejecutar siempre los procedimientos de separación logra compensar esta situación, y se obtiene un tiempo de ejecución levemente superior al normal (en promedio 1.2 veces mayor).

Por último, la estrategia de "generación probabilistica" logra tiempos muy cercanos a los normales, a pesar de su dudosa apariencia. El promedio de los tiempos resulta sólo un 10% superior al obtenido aplicando todos los cortes en la segunda fase. Las probabilidades utilizadas en esta estrategia fueron proporcionales a las frecuencias de las desigualdades (tabla 7.3), pero podría experimentarse con otras distribuciones de probabilidad.

7.4 Comparación con el algoritmo *branch&bound*

Si solamente consideramos el esquema de enumeración de un algoritmo *branch&bound*, tendremos un método exacto para resolver el problema, pero su tiempo puede ser muy grande. Estos tiempos pueden mejorarse si se incluye la generación de cortes en cada nodo del árbol, en cuyo caso tendremos un algoritmo *branch&cut*. Se espera que, a pesar del tiempo adicional de los procedimientos de separación, las desigualdades que se agreguen a la formulación permitan reducir el tiempo total de ejecución, al obtenerse árboles más pequeños.

Una segunda mejora puede obtenerse mediante la inclusión de una heurística primal. En las pruebas descriptas en las secciones anteriores, no se utilizaron heurísticas primales, para evaluar solamente el rendimiento del proceso *branch&cut*. Sin embargo, esta heurística está presente en la implementación final, dado que permite disminuir los tiempos de ejecución.

En las secciones anteriores, se realizaron pruebas para analizar la calidad empírica de las desigualdades válidas, y para ajustar los parámetros del algoritmo. Arribamos en esta sección a la primera comparación entre métodos distintos. En las pruebas que se resumen en la tabla 7.10, se ejecutaron estos tres métodos (*branch&bound* y *branch&cut* con y sin heurística primal) sobre el conjunto de problemas test. Para cada método, se muestra el tiempo total de ejecución (en sg.) y la cantidad de nodos del árbol de enumeración. Se indica con "***" las instancias que el algoritmo *branch&bound* no pudo resolver, al agotar la memoria virtual existente (aproximadamente 130 MB en todas las pruebas).

Debe notarse que se realiza una comparación entre los distintos métodos, y no entre distintas implementaciones. Se utilizó en todos los casos una implementación sobre el entorno ABACUS, que representa como objetos en memoria todos los elementos del proceso (nodos del árbol, variables, restricciones, etc.). Esta flexibilidad del entorno tiene el costo de no proporcionar un manejo eficiente de la memoria (dado que queda bajo la responsabilidad

	Branch&Bound		Branch&Cut		Heurist. Primal	
Problema	Tiempo	#Subpr.	Tiempo	#Subpr.	Tiempo	#Subpr.
DC2	***	***	1594.80	1785	0.53	1
DF1	***	***	1774.22	1363	496.23	233
PRB3	0.10	9	0.06	1	0.06	1
PRB4	0.31	31	0.23	15	0.09	1
PRB5	19.98	703	2.36	37	0.19	1
PRB6	72.84	1669	0.45	5	0.19	1
PRB7	385.68	6769	60.62	205	0.40	1
PRB7B	20.89	561	5.17	63	0.24	1
PRB8	***	***	475.31	961	25.11	55
PRB8B	187.26	5173	1.48	31	0.16	1
PRB9	***	***	122.68	255	0.73	1
PRB9B	61.38	1369	100.35	727	0.21	1
STR1	***	***	67.80	257	0.94	1
STR2	38.03	1301	0.64	17	0.15	1
WARS1	2.34	121	1.56	27	0.14	1
WARS2	2.40	133	3.18	63	0.13	1
WARS3	70.45	1615	15.59	129	2.61	15
WARS4	***	***	1.23	31	0.21	1
GAUSS1	***	***	856.24	1551	542.14	915

Tabla 7.10: Comparación entre B&B, y B&C con y sin heurística primal.

del lenguaje), con lo cual soporta árboles de menor tamaño que, por ejemplo, el paquete CPLEX. Por estos motivos, la implementación del algoritmo *branch&bound* agota la memoria en instancias medianas, lo cual no sucede con CPLEX. Sin embargo, no es el objetivo de esta sección la comparación contra este paquete, que se realiza más adelante.

Como puede verse en la tabla 7.10, existe una notable diferencia entre cada uno de los métodos. El algoritmo *branch&bound* puede resolver gran parte de los problemas, mientras que al agregar cortes se resuelve la totalidad de las instancias, con tiempos mucho menores en casi todos los casos, y logrando árboles más pequeños en todos los casos. En algunas instancias, la reducción del tamaño del árbol es dramática (de 1669 a 5 nodos para el problema PRB6), y la ganancia de tiempo muy significativa (de 187 sg. a 1 sg. para PRB8B). La inclusión de los procedimientos de separación demuestra entonces un gran impacto sobre el tiempo de ejecución.

Sólo en dos casos el agregado de planos de corte provoca tiempos mayores (problemas PRB9B y WARS2). Debe notarse que en ambos casos los árboles tienen una menor cantidad de nodos en el método *branch&cut*, al incorporarse los planos de corte. Es decir, los cortes contribuyen a disminuir el tamaño del árbol, pero la ejecución de los procedimientos de separación supone un tiempo adicional que no es compensado por la reducción del árbol.

Se observa un salto similar cuando se agrega la heurística primal. Los tiempos de ejecución vuelven a mejorar en gran medida, dado que ahora se tiene una heurística que proporciona

una buena solución inicial, cuyo valor ayuda a podar el árbol. En muchos casos, se redujo el árbol de enumeración a *un sólo nodo*, al hallar el óptimo la heurística y los planos de corte simultáneamente[2]. En los casos en los cuales no se obtuvo el óptimo en el primer nodo, el tamaño del árbol disminuyó sensiblemente (por ejemplo, de 1363 a 263 nodos para el problema DF1).

En resumen, estas pruebas muestran los dos grandes "saltos" en el tiempo de ejecución. Se obtienen tiempos mucho mejores cuando se agrega la generación de cortes al algoritmo *branch&bound*, y se vuelven a mejorar sustancialmente cuando se incorpora la heurística primal. Si se comparan ahora los resultados de la primera columna contra la última, pueden observarse mejoras de hasta tres ordenes de magnitud en el tiempo de ejecución y la cantidad de nodos del árbol de enumeración.

7.5 Comparación con CPLEX

Hasta el momento, el único medio para intentar resolver el problema de *mapping* en forma exacta consistía en formular su modelo de programación lineal entera y aplicar algún paquete de optimización. El paquete CPLEX es una de las mejores implementaciones existentes para la resolución de problemas de programación lineal y programación lineal entera (cfr. [Cpl/94]). Una vez realizadas las pruebas de las secciones anteriores, y ajustados los parámetros del algoritmo, se realizaron comparaciones de los tiempos de ejecución del algoritmo *branch&cut* y el paquete CPLEX 4.0. Debe notarse que el entorno ABACUS utiliza la misma versión de CPLEX para la resolución de los subproblemas lineales, con lo cual la comparación se realiza "en igualdad de condiciones".

En la tabla 7.11 se muestran los resultados de las pruebas para un conjunto representativo de problemas. Se indica para cada método la garantía de optimalidad alcanzada (que es 0% si se encontró el óptimo), el tiempo total de ejecución en segundos y el número de nodos del árbol de enumeración. Se ejecutaron las pruebas en una misma máquina con un límite de tiempo de una hora de CPU, y en los casos en los cuales no se pudo encontrar el óptimo en este lapso, se indica "*****" en la segunda columna.

En primer lugar, se puede observar que el algoritmo *branch&cut* obtiene mejores tiempos en casi todos los problemas, logrando diferencias dramáticas en algunos casos. Algunos problemas pueden ser resueltos por el primer algoritmo pero no logran resolverse dentro del límite de tiempo impuesto utilizando CPLEX (problemas STR1, STR8 y PRB8, entre otros).

En aquellos problemas que no pudieron ser resueltos por ninguno de los dos métodos (últimas filas de la tabla), la garantía de optimalidad[3] arrojada por el algoritmo *branch&cut* es mucho mejor. La heurística primal incorporada a este algoritmo proporciona una solución que es, en general, de buena calidad. Por otra parte, los planos de corte dados por las

[2] En realidad, esto sucede también cuando al generar cortes para el primer nodo se obtiene una solución cuyo valor es igual al óptimo, aunque no sea entera.

[3] Si las cotas inferior y superior al momento de llegar al límite de tiempo son z_{IP} y z_{LP}, entonces la garantía de optimalidad se calcula como $gap = 100\frac{z_{LP}-z_{IP}}{z_{IP}}$ (cfr. [Aar/95b])

	branch&cut			CPLEX 4.0		
Problema	Garantía	Tiempo	Nodos	Garantía	Tiempo	Nodos
PRB5	0.00%	0.19	1	0.00%	1.42	106
PRB6	0.00%	0.19	1	0.00%	8.88	766
PRB7	0.00%	0.40	1	0.00%	243.57	11800
PRB7B	0.00%	0.24	1	0.00%	2.96	225
STR1	0.00%	0.94	1	71.78%	*****	162576
STR2	0.00%	0.15	1	0.00%	3.74	1555
STR8	0.00%	117.93	1	392.80%	*****	513
STR9	0.00%	180.98	1	444.00%	*****	1036
STR10	37.59%	*****	269	387.50%	*****	3021
STR11	9.52%	*****	111	283.32%	*****	951
STR17	0.00%	328.98	1	413.65%	*****	717
STR19	4.34%	*****	69	437.09%	*****	507
WARS1	0.00%	0.14	1	0.00%	0.05	3
WARS2	0.00%	0.13	1	0.00%	0.05	5
WARS3	0.00%	2.61	15	0.00%	2.42	209
WARS4	0.00%	0.21	1	0.00%	161.55	46680
WARS5	0.00%	1.79	7	0.00%	7.69	1113
PRB8	0.00%	25.11	55	71.45%	*****	103829
PRB8B	0.00%	0.16	1	0.00%	64.79	17342
DF1	0.00%	496.23	233	40.00%	*****	69577
DF2	10.00%	*****	143	328.57%	*****	1953
DF3	23.80%	*****	737	130.76%	*****	17669
DF4	30.00%	*****	1009	98.26%	*****	26987
DF5	11.53%	*****	521	177.77%	*****	12702
DF6	7.40%	*****	813	38.04%	*****	17466
DF7	10.34%	*****	463	263.63%	*****	9728
DF8	6.66%	*****	599	206.38%	*****	13780

Tabla 7.11: Comparación entre el algoritmo B&C y CPLEX 4.0

familias de desigualdades válidas contribuyen a disminuir el óptimo de la relajación lineal, reduciendo además el tamaño del árbol. La conjunción de la heurística primal con la acción de los procedimientos de separación produce estos resultados, que se traducen en garantías más pequeñas. Es decir, el primer algoritmo logra acotar el óptimo del problema dentro de un rango mucho más reducido que el obtenido por CPLEX.

Un caso más drástico se produjo para los problemas detallados en la tabla 7.12. En estos problemas, el proceso B&B implementado por CPLEX no pudo hallar ninguna solución factible entera luego de una hora de ejecución, con lo cual se tiene sólo una cota superior del óptimo. Esta tabla muestra las cotas inferior y superior logradas por el algoritmo *branch&cut* (se indica entre paréntesis la garantía que conforman), su tiempo de resolución y la cota superior lograda por CPLEX. Debe notarse que el algoritmo B&C pudo resolver los dos primeros problemas, y que la garantía en los tres restantes no supera el 10%.

Problema	Cotas Sup./Inf. B&C	Tiempo B&C	Cota Sup. B&B
STR14	61/61 (0.00%)	941.15	142.18
STR15	59/59 (0.00%)	418.65	119.41
STR16	51/54 (5.88%)	*****	118.64
STR18	47/50 (6.38%)	*****	119.12
STR20	62/64 (3.22%)	*****	138.77

Tabla 7.12: Instancias en las que CPLEX no halló solución factible.

La tabla 7.11 muestra los tiempos de resolución, pero también es de interés evaluar la cantidad de nodos que presentan los árboles de enumeración para cada método. Consideremos, por ejemplo, el problema WARS3. Los tiempos de resolución de los dos algoritmos son similares (2.61 sg. para el primero y 2.42 sg. para el segundo), pero mientras el algoritmo B&C recorre 15 nodos, el paquete CPLEX genera un árbol de 209 subproblemas, es decir, más de 13 veces mayor. Esta importante diferencia se debe a la excelente implementación de CPLEX, que logra una performance altamente eficiente[4]. En todos los casos, la cantidad de nodos generados por unidad de tiempo es muy superior en CPLEX, a pesar de lo cual el algoritmo *branch&cut* logra tiempos menores.

Es decir, la reducción del tiempo de ejecución que aportan la heurística primal y los planos de corte logra tiempos menores, a pesar de que la implementación no tenga la misma eficiencia de CPLEX. De todos modos, el objetivo principal del trabajo de programación fue verificar la fuerza de las desigualdades halladas, y testear la efectividad de sus procedimientos de separación. Se obtuvo un algoritmo muy competitivo para el problema, pero una implementación más cuidada podría lograr tiempos mucho más bajos. Debe notarse que un programa de estas características debería encargarse del manejo del árbol de enumeración, la generación (y quizás también la resolución) de los subproblemas lineales, y el proceso *branch&bound*, dado que por razones de eficiencia no podría utilizar el entorno ABACUS. De esta forma, se pierde la flexibilidad de este entorno, haciendo que esta implementación sea un proyecto de gran envergadura.

Conclusiones. La comparación entre el algoritmo *branch&cut* y el paquete CPLEX muestra que el primero es una buena alternativa para la resolución del problema. A pesar de la notable calidad y la gran eficiencia de la implementación de CPLEX, las desigualdades halladas (junto con sus procedimientos de separación) combinadas con el uso de una buena heurística primal, constituyen una mejora muy importante al método *branch&bound*, que logra tiempos y garantías mucho menores.

[4] El algoritmo *branch&cut* se construyó completamente sobre un lenguaje de alto nivel orientado a objetos (con el costo extra que suponen los mecanismos propios de estos lenguajes) y utilizando un entorno cuya implementación no puede modificarse. Este hecho, sumado al costo de las heurísticas y procedimientos de separación, hace que el algoritmo B&C recorra una cantidad menor de nodos en el mismo tiempo.

7.6 Resultados computacionales adicionales

Las pruebas descriptas en las secciones anteriores se realizaron sobre un conjunto reducido de instancias, por las razones que se indicaron en la sección 7.1.1. La tabla 7.14 resume los resultados obtenidos para todas las instancias de prueba mencionadas en este anexo, realizando una comparación con el paquete CPLEX 4.0. Para cada algoritmo, se indica el tiempo total de ejecución en sg. (con un límite de tiempo de una hora), la cantidad de nodos del árbol de enumeración y la garantía de optimalidad alcanzada (se indican con asteriscos los casos en los cuales no se pudieron obtener soluciones factibles).

Como puede verse, los resultados obtenidos son similares a los de la tabla 7.11 de la pagina 127, lo cual confirma las conclusiones a las que se arribó en las secciones precedentes.

	branch&cut			CPLEX 4.0		
Problema	Tiempo	#Subpr.	Garantía	Tiempo	#Subpr.	Garantía
DC1	0.74	1	0.00%	9.15	42	0.00%
DC2	0.66	1	0.00%	6.13	51	0.00%
DC3	*****	155	11.11%	*****	2109	11.11%
DC4	*****	117	3.45%	*****	1276	25.00%
DF1	537.11	321	0.00%	*****	62695	37.50%
DF2	*****	165	12.82%	*****	1905	328.57%
DF3	*****	781	23.81%	*****	17379	130.77%
DF4	*****	1223	30.00%	*****	27578	93.33%
DF5	*****	647	11.54%	*****	11186	288.89%
DF6	*****	1069	7.41%	*****	16045	36.00%
DF7	*****	581	10.34%	*****	9397	263.64%
DF8	*****	663	6.67%	*****	12818	200.00%
DF11	480.66	499	0.00%	*****	89573	11.11%
DF12	*****	2881	4.35%	1410.21	28371	0.00%
DF13	*****	3647	8.70%	1575.98	31121	0.00%
DF14	*****	641	30.43%	*****	11553	100.00%
DF15	*****	871	22.22%	*****	9286	94.44%
DF16	*****	911	20.00%	*****	7398	90.48%
DF17	*****	945	16.13%	*****	7546	90.48%
DF18	*****	747	18.18%	*****	5790	95.65%
DF19	*****	451	16.67%	*****	3324	108.33%
DF20	*****	445	18.42%	*****	2407	129.17%
GAUSS1	542.14	915	0.00%	185.39	2109	0.00%
GAUSS2	*****	715	4.55%	*****	10858	47.06%
GAUSS3	*****	811	4.55%	*****	21099	14.29%
GAUSS4	*****	847	9.52%	*****	12810	38.89%
MEMSY1	*****	631	10.71%	*****	2386	166.67%
MEMSY2	*****	379	18.75%	*****	1295	90.48%
MEMSY3	*****	853	9.09%	*****	3711	100.00%
MEMSY4	*****	439	20.69%	*****	1589	71.43%
MEMSY5	*****	641	16.00%	*****	2817	114.29%
MEMSY6	13.83	1	0.00%	*****	3105	71.43%
MEMSY7	*****	897	8.00%	*****	2343	80.00%
MEMSY8	*****	875	9.09%	*****	3225	84.62%
PRB1	0.07	1	0.00%	<0.00	1	0.00%
PRB10	*****	1649	10.53%	*****	4774	100.00%
PRB2	0.05	1	0.00%	0.03	11	0.00%
PRB3	0.08	1	0.00%	0.02	6	0.00%
PRB4	0.09	1	0.00%	0.02	11	0.00%
PRB5	0.17	1	0.00%	1.39	106	0.00%
PRB6	0.22	1	0.00%	9.10	766	0.00%

Tabla 7.13: Resultados para todas las instancias de prueba

	branch&cut			CPLEX 4.0		
Problema	Tiempo	#Subpr.	Garantía	Tiempo	#Subpr.	Garantía
PRB7	0.53	1	0.00%	263.52	11800	0.00%
PRB7B	0.20	1	0.00%	3.10	225	0.00%
PRB8	17.50	47	0.00%	*****	94579	63.64%
PRB8B	0.17	1	0.00%	71.18	17342	0.00%
PRB9	0.86	1	0.00%	310.21	10976	0.00%
PRB9B	0.25	1	0.00%	5.83	501	0.00%
STR1	1.04	1	0.00%	*****	141259	71.43%
STR2	0.15	1	0.00%	3.98	1555	0.00%
STR3	*****	1041	4.00%	*****	9507	250.00%
STR4	*****	953	4.00%	*****	11982	205.56%
STR5	53.99	1	0.00%	*****	2855	234.78%
STR6	*****	371	2.78%	*****	2375	310.53%
STR7	*****	153	2.13%	*****	514	326.09%
STR8	117.93	1	0.00%	*****	510	390.00%
STR9	180.98	1	0.00%	*****	1030	438.89%
STR10	*****	369	2.78%	*****	2651	387.50%
STR11	*****	175	4.35%	*****	882	280.00%
STR12	*****	319	8.82%	*****	1856	247.83%
STR13	*****	191	8.57%	*****	1033	438.89%
STR14	844.01	1	0.00%	*****	111	*****
STR15	*****	69	9.26%	*****	19	*****
STR16	*****	65	8.00%	*****	56	*****
STR17	566.57	1	0.00%	*****	656	413.04%
STR18	*****	81	6.38%	*****	52	*****
STR19	*****	69	4.34%	*****	495	436.36%
STR20	1153.09	1	0.00%	*****	31	*****
WARS1	0.13	1	0.00%	0.05	3	0.00%
WARS2	0.16	1	0.00%	0.05	5	0.00%
WARS3	2.61	15	0.00%	2.40	209	0.00%
WARS4	0.17	1	0.00%	171.35	46680	0.00%
WARS5	1.79	7	0.00%	8.09	1113	0.00%

Tabla 7.14: Resultados para todas las instancias de prueba (cont.)

Capítulo 8

Conclusiones

"Y yo estaré siempre con ustedes hasta el fin del mundo."

–Mt., 28. 20

A lo largo de los capítulos precedentes se mostró el camino recorrido para la implementación de un algoritmo *branch&cut* para el problema de *mapping* simple. Se demostró que este problema pertenece a la clase de problemas *NP-Hard*, y que sucede lo mismo con los subproblemas que aparecen con frecuencia en la práctica. Este hecho, frecuente en los problemas de optimización combinatoria, obliga a orientar la búsqueda (a) hacia los métodos heurísticos, o bien (b) hacia algoritmos exactos pero con peor caso exponencial. A este segundo grupo pertenecen los algoritmos *branch&cut*, y la implementación de un algoritmo de este tipo fue uno de los objetivos principales del presente trabajo.

En el capítulo 3 se describieron dos métodos heurísticos implementados para este problema, ambos consistentes en adaptaciones de técnicas metaheurísticas. El segundo de ellos (CLO) se reveló muy eficiente y robusto en la práctica, y fue capaz de alcanzar el óptimo de varios problemas en tiempos cortos. Este método presenta similitudes con la heurística presentada en [Bok/81], sobre la cual también se reportan buenos resultados, en base a la tecnología existente al momento de su implementación. La diferencia entre este artículo y el presente informe radica en que gracias al algoritmo *branch&cut*, ahora se conocen los óptimos de varios problemas (o al menos se tienen cotas que lo "encierran" en intervalos pequeños), y se puede evaluar la calidad de la heurística con relación a estos datos.

El estudio poliedral del problema, descripto en el capítulo 5, constituye el núcleo fundamental del algoritmo. En este estudio inicial, se formularon dos modelos de programación lineal entera equivalentes, y se encontró la dimensión del poliedro asociado con ellos. Este hecho es muy importante, dado que permitió conocer la calidad de las desigualdades, dada por la dimensión de la cara que definen. En este sentido, se halló un conjunto de desigualdades válidas, entre las cuales se encuentran tres familias exponenciales y una familia polinomial de facetas. Muchas de estas desigualdades se incorporaron al algoritmo, recurriendo a procedimientos de separación heurísticos y, en algunos casos, exactos (es decir, que identifican una desigualdad violada en la familia o garantizan que ninguna lo está).

Debe notarse que en muchos casos, especialmente en problemas de *scheduling*, no se conoce la dimensión de los poliedros asociados con los problemas (cfr. por ejemplo [Mac/95] y [Jun/96]). Resultó de gran importancia conocer la dimensión del poliedro del problema de *mapping*, dado que esta información fue decisiva para probar que algunas familias definen facetas.

Una de las posibilidades que se consideraron cuando se formuló el modelo del problema fue agregar restricciones para forzar que las variables de costo c_{ij} valieran 1 cuando i y j se asignaran a procesadores conectados (en el modelo actual, esta variable puede valer 0 o 1 en este caso, y está obligada a valer 0 en caso contrario). La razón más importante para no incluir estas restricciones fue la dificultad que ocasionaron para hallar la dimensión del nuevo poliedro, complicando enormemente el análisis de la dimensión de las caras definidas por las desigualdades: Al no conocer la dimensión del poliedro, debe probarse que definen facetas por medios indirectos, que pueden resultar muy complejos.

Como se mencionó, se pudo probar que algunas familias de desigualdades definen facetas del poliedro, incluyendo a tres familias exponenciales. Las familias exponenciales de desigualdades (especialmente de facetas) son muy importantes (cfr. [Mag/97]), al punto que en algunas implementaciones sólo se hace uso de estas familias (cfr. [Jun/92]). En esta implementación se utilizaron medios heurísticos para su separación, pero sería de interés indagar la existencia de procedimientos exactos, al menos para un subconjunto de estas desigualdades.

Entre las familias de desigualdades, los experimentos computacionales revelaron un subconjunto de mayor trascendencia, que es responsable de buena parte de las mejoras en la performance del proceso *branch&bound*. Como se describió en el capítulo 7, este núcleo está formado por casi todas las familias exponenciales, y constituye uno de los puntos más importantes del presente trabajo. Al identificar (empíricamente) un grupo de familias gracias al cual se logran los mayores avances, se deja sentado un precedente de interés para futuros estudios sobre el problema.

Otro punto importante que mostraron las pruebas computacionales fue la mejora sobre el tiempo de ejecución y el número de nodos del árbol de enumeración que aporta la generación de las desigualdades válidas halladas. La implementación del algoritmo comenzó con un conjunto reducido de desigualdades, al cual se fueron agregando progresivamente nuevas familias. Resultó muy notable observar los tamaños de los problemas que el algoritmo pudo resolver a medida que se incorporaron nuevas familias. En una implementación preliminar, problemas de 10 nodos eran prácticamente irresolubles, mientras que en las últimas versiones del algoritmo estos problemas pudieron resolverse en décimas de segundo, siendo necesario sólo un nodo del árbol.

En este sentido, muchos problemas pequeños pudieron resolverse en un solo nodo, dado que la heurística primal pudo hallar el óptimo, y en la primera generación de cortes se logró una relajación con este mismo valor. Esto es muy positivo a la hora de evaluar la fuerza de las desigualdades válidas. Por otra parte, en los problemas que no pudieron resolverse en tan corto lapso, las desigualdades cumplen un papel fundamental para detener la explosión del tamaño del árbol de enumeración. El gráfico 8.1 muestra la cantidad de nodos abiertos del árbol a medida que pasan las iteraciones, para el problema DF1: Nótese cómo esta cantidad

aumenta vertiginosamente al comienzo (dos nuevos nodos por iteración), pero luego la acción de las desigualdades hace que las relajaciones tengan óptimos menores, haciendo descender la cota superior y podando así grandes ramas del árbol.

Figura 8.1: Cantidad de nodos abiertos en función del tiempo.

Las desigualdades válidas, en especial el núcleo principal de familias exponenciales, producen el primer gran "salto" en los tiempos de ejecución. Como se mostró en el capítulo anterior, los tiempos del algoritmo *branch&bound* inicial se mejoran dramáticamente cuando se incorporan las desigualdades. El segundo salto se produjo al agregar la heurística primal inicial, que proporciona una cota inferior del óptimo. Nuevamente, la disminución de los tiempos es importante y es en este punto cuando se logran resoluciones con un sólo nodo en el árbol.

Debe observarse, de todos modos, que los problemas que se pudieron resolver son de tamaño pequeño y mediano. El problema más pequeño que no se pudo resolver tiene 15 nodos (para STR4 se llegó a una garantía de 4.16%), y el problema más grande que se pudo resolver tiene 35 nodos. Esta situación se explica por dos motivos. En primer lugar, el algoritmo se basa sobre un estudio inicial del poliedro. Las desigualdades incorporadas al algoritmo son exclusivamente producto de este trabajo, dado que no tenemos conocimiento de estudios similares anteriores sobre este mismo problema. Las implementaciones de algoritmos *branch&cut* más espectaculares se basan en los trabajos de muchos autores, y suponen un conocimiento profundo de la estructura de los poliedros asociados con los problemas. En este sentido, el algoritmo aquí presentado constituye una implementación experimental, realizada para resolver efectivamente instancias del problema, pero también pensada como un "laboratorio computacional" para testear la fuerza empírica de las desigualdades halladas.

En segundo término, debe realizarse una observación sobre el tamaño del modelo para este problema. La tabla 8.1 muestra la cantidad de variables binarias de los modelos de varios problemas, en función del tamaño de la instancia. Como puede verse, el modelo asociado con el problema de *mapping* es muy grande, superando las 250 variables binarias para instancias

pequeñas. Esto sucede dado que *las soluciones factibles del problema son las biyecciones entre dos conjuntos* (los conjuntos de nodos de los dos grafos), y deben representarse en forma adecuada. Debe notarse que [Jun/94] califica la resolución de problemas de 100 variables binarias como "una tarea no trivial". La confluencia de estos dos factores produce los resultados obtenidos en cuanto al tamaño de los problemas resueltos.

Tamaño instancia		Variables binarias			
Nodos	Ejes	TSP	Steiner	Clique	*Mapping*
5	10	10	≤ 15	5	35
10	15	15	≤ 30	10	115
10	25	25	≤ 35	10	125
15	30	30	≤ 45	15	255
15	60	60	≤ 75	15	285
20	30	30	≤ 60	20	430
20	50	50	≤ 70	20	430
30	50	50	≤ 80	30	950
30	70	70	≤ 100	30	970

Tabla 8.1: Número de variables binarias de los modelos de distintos problemas.

Debe mencionarse también, aunque esto no conduzca necesariamente a concluir su dificultad, que este problema es un subproblema de QAP, que se cuenta entre los de más difícil resolución. Hasta la fecha, la mayor instancia resuelta en la biblioteca de instancias QAPLIB para este problema tiene 32 nodos, pero existen instancias de 20 y hasta 15 nodos aún sin resolver (cfr. [Bur/91]).

De todos modos, y teniendo en cuenta el carácter experimental del algoritmo implementado, se obtuvieron tiempos de resolución superiores al paquete CPLEX. Hasta la fecha, la única forma de intentar la resolución exacta de este problema consistía en formular su modelo de programación lineal entera y utilizar este paquete u otros similares. El algoritmo *branch&cut* implementado permitió resolver instancias que CPLEX no pudo resolver, y obtuvo mejores cotas del óptimo en todos los casos, dado un límite fijo de tiempo.

Por último, es preciso hacer una mención al problema dual descripto en el capítulo 4. Como se indicó anteriormente, este problema dual surgió con la necesidad de hallar cotas superiores para el óptimo, y aunque presenta interés teórico, no resultó adecuado en la práctica para generar buenas aproximaciones a este valor. Otro enfoque que encuentra el óptimo acercándose progresivamente con cotas superiores está dado por un algoritmo basado en la reformulación de Benders (cfr. [Mag/81] y [Mag/86]). Se logró aplicar la reformulación de Benders al problema y se implementó efectivamente una versión preliminar de este algoritmo, pero sus resultados no fueron buenos. Logró resolver instancias pequeñas, pero utilizando tiempos mucho mayores que el algoritmo *branch&cut*, y para instancias más grandes, las cotas superiores provistas se ubicaron a mucha distancia del óptimo de los problemas.

En conclusión, de los tres enfoques intentados para la generación de cotas superiores del problema (problema dual, reformulación de Benders y planos de corte), debe señalarse que este último logró los mejores resultados. Para el problema de *mapping*, el estudio del poliedro asociado al problema y la incorporación de las desigualdades halladas en un algoritmo

branch&cut es el camino que se muestra más propicio para su resolución.

8.1 Trabajos futuros

El estudio realizado constituye un primer intento de conocer la estructura del poliedro asociado con este problema. Sería muy interesante una continuación de estos estudios, para buscar nuevas desigualdades válidas y familias de facetas. En este sentido, las intuiciones logradas a lo largo de este trabajo pueden ser de utilidad para esta continuación. Puede ser también de interés la consideración de modelos de programación lineal entera alternativos para este mismo problema.

Con relación a las desigualdades válidas halladas, puede ser de gran importancia la definición de nuevos procedimientos de separación. Es posible que para algunas de las desigualdades existan procedimientos de separación exactos, que constituirían un gran aporte al algoritmo. Pueden considerarse también otros procedimientos heurísticos, que deberían someterse a pruebas computacionales para determinar su eficacia.

Otra posibilidad consiste en estudiar problemas relacionados, con funciones objetivo distintas e inclusive con tiempos sobre las tareas. Un enfoque híbrido podría dejar de considerar a las tareas como simultáneas, teniendo en cuenta su duración y tiempos de comienzo. Un problema de estas características se asemeja a los problemas habituales de *scheduling*, con lo cual la experiencia ganada en este ámbito puede ser muy valiosa.

Por último, pueden buscarse nuevos problemas duales, especialmente considerando los recientes intentos sobre duales de QAP (cfr. [Hah/98] y [Res/95]). Un dual para este problema es también un dual para el problema de clique máxima, con lo cual tiene gran interés teórico. Pueden intentarse modificaciones al dual hallado, e inclusive es posible que existan procedimientos para disminuir su tamaño (se realizaron intentos en este sentido, pero no se lograron reducciones importantes). Otra posibilidad es intentar por caminos distintos, que conduzcan a duales de otras características. En cualquier caso, un problema dual aplicable a instancias prácticas sería realmente importante.

***

El reciente progreso en la resolución exacta de problemas de optimización combinatoria por medio de algoritmos de tipo *branch&cut* y el consecuente interés que despierta esta actividad proporcionan una gran motivación para multiplicar los esfuerzos en esta disciplina. El presente trabajo representa un intento en esta dirección, al iniciar el estudio poliedral del problema de *mapping* simple, que representa los primeros pasos para su resolución. Aún hallándose a gran distancia de los espectaculares avances logrados en otros problemas de optimización, el conjunto de desigualdades válidas (entre ellas, las facetas) encontradas para este poliedro y el algoritmo al cual se incorporan constituyen un aporte para la resolución del problema.

Apéndice A

Descripción de la implementación

"Y la ola incidente era tan intensa, la necesidad de subir la gran montaña tan imperiosa, que finalmente apareció en la Tierra un ser en cuya mente había una sospecha de lo que ocurría."

–Fred Hoyle

Se describe en este anexo la implementación del algoritmo *branch&cut* en detalle. En primer lugar, se mencionan las características del *framework* utilizado, para pasar luego a los aspectos fundamentales del algoritmo. Por último, se describe la implementación de los refinamientos al algoritmo, mencionados en el capítulo 6 (estrategia de *branching* modificada, fijado de variables por implicaciones lógicas, etc.).

A.1 El entorno ABACUS

El *framework* ABACUS (A Branch-And-CUt Software) fue escrito por Stefan Thienel como parte de su tesis doctoral, dirigida por Michael Jünger (cfr. [Thi/95]). Este entorno es un *framework* orientado a objetos para la implementación de algoritmos *branch&cut*, cuyo diseño tiene especialmente en cuenta los problemas de optimización combinatoria. Está escrito en el lenguaje de programación C++ y se organiza como un conjunto de clases, que implementan un algoritmo *branch&cut* general. El esquema de herencia y redefinición de funciones soportado por este lenguaje permite que el entorno provea "puntos de entrada" en los cuales se ejecutan funciones específicas definidas por el usuario, como se describe en las próximas secciones.

A continuación, se describen las características principales de este entorno (cfr [Jun/97]).

- **Método implementado.** Este entorno implementa un algoritmo *branch&cut* general basado en relajaciones lineales. El manejo del árbol de enumeración, los subproblemas y el proceso de *branching* quedan a cargo del entorno. El usuario puede agregar cortes

durante el proceso de optimización, implementando sus propios procedimientos de separación, que pueden basarse en conceptos propios del problema a resolver.

- **Representación del problema.** El problema lineal entero se representa por medio de restricciones, junto con la función objetivo. Estas restricciones se implementan como subclases de una clase especial de ABACUS, definidas por el usuario. De esta forma, el usuario puede representar las restricciones con conceptos del problema (cuidando de respetar la interface solicitada por el entorno), simplificando la implementación y evitando el manejo de índices durante la generación de cada fila de la matriz del problema lineal. Sucede lo mismo con las variables.

- **Resolución de los problemas lineales.** ABACUS no implementa directamente la resolución de los problemas lineales que aparecen durante la optimización, sino que deja esta tarea a cargo de programas especializados. Actualmente, soporta interfaces con CPLEX, SOPLEX y XPRESS. La interface con estos programas se realiza en forma automática.

- **Manejo de la memoria.** Las restricciones y las variables del problema, junto con los cortes generados a lo largo del proceso se guardan en *pools* especiales, manejados por el entorno. Es posible utilizar *pools* específicos para propósitos particulares, como se realiza para implementar la estrategia de *branching* especializada para este problema.

- **Estrategias.** El entorno soporta distintas estrategias de selección de la variable de *branching* (en el caso de *branching* por variables), fijado de variables, recorrido del árbol de enumeración, frecuencia de la separación, etc. Adicionalmente, se pueden agregar estrategias específicas para cada problema, que el entorno soporta proporcionando "puntos de entrada" para las funciones que las implementan.

- **Manejos de parámetros.** El proceso tiene asociado un gran número de parámetros, y existen dos formas de modificarlos. Por una parte, se tiene un archivo con los valores de los parámetros, que se lee al comienzo del proceso. Por otra parte, estos parámetros pueden cambiarse durante la ejecución mediante funciones especiales, provistas por el entorno.

A.1.1 La estructura de clases de ABACUS

Este entorno está organizado como un conjunto de clases, que implementan distintos aspectos del proceso. Entre ellas, cuatro son de importancia para la implementación de algoritmos *branch&cut*, y con ellas el usuario de ABACUS debe interactuar. El usuario no puede modificar la implementación de estas clases, sino que debe construir subclases derivadas de ellas, que son las que realmente se instanciarán.

Clase `MASTER`

Esta clase[1] se instancia en un sólo objeto, que controla el proceso de optimización. Entre otras tareas, inicializa el árbol de enumeración, genera el primer subproblema y se encarga de controlar el proceso *branch&bound*. La generación de las variables y restricciones del primer subproblema queda a cargo del usuario, mientras que el entorno lleva a cabo los restantes procedimientos.

Tres métodos de esta clase son de gran importancia. En primer lugar, debe mencionarse al **constructor** de la clase, que habitualmente recibe el nombre de un archivo del cual lee los datos del problema. Por otra parte, el método `optimize()` inicia el proceso de optimización, llamando entre otras a la función `initializeOptimization()`, que debe inicializar las variables y restricciones de la formulación del subproblema inicial. Debe observarse que esta clase realiza el manejo de parámetros, que se leen de un archivo cuyo nombre se fija de antemano (en este caso, se mantuvo el nombre `.abacus` predefinido). Los parámetros permiten controlar todo el proceso, y pueden modificarse directamente en este archivo, o mediante las funciones que esta clase provee.

Clase `SUB`

Esta clase representa cada subproblema del árbol de enumeración, y se instancia una vez por cada nodo (cada objeto de esta clase representa un nodo del árbol). Tiene dos constructores, que corresponden al nodo inicial y a los restantes nodos, respectivamente. El primero de ellos solamente recibe un puntero al objeto `MASTER`, del cual obtiene la información necesaria para comenzar el proceso. El segundo constructor recibe además un puntero a su nodo padre (un objeto de clase `SUB`) y una regla de *branching* (un objeto de clase `BRANCHRULE`).

Cada objeto de esta clase resuelve el subproblema que tiene asociado, genera cortes para la solución hallada, y repite el proceso tantas veces como indique el parámetro correspondiente. Si el nodo puede cerrarse, se elimina de la lista de nodos abiertos, pero si esto no sucede, genera nuevos hijos por *branching*, que serán nuevos objetos de esta misma clase, y que se agregan a la lista de nodos abiertos.

Por último, debe mencionarse que `SUB` provee diferentes puntos de entrada al proceso de optimización, para que el usuario pueda intercalar sus propios procedimientos. Estos puntos se implementan como funciones que no realizan ninguna acción, pero que pueden redefinirse en la subclase derivada de `SUB` para que ejecuten las tareas indicadas por el usuario. Los cuatro puntos de entrada permiten adicionar (a) procedimientos de separación para generar planos de corte, (b) *pricing* de variables, (c) ejecución de heurísticas en cada nodo y (d) fijado de variables por implicaciones lógicas.

[1] Las clases tienen el prefijo `ABA` delante de su nombre, con lo cual `MASTER` se llama en realidad `ABA_MASTER`, pero en la literatura sobre ABACUS se omite en general este prefijo por claridad en la exposición. Sucede lo mismo con las otras clases.

Clase `VARIABLE`

Cada variable es un objeto de esta clase. El usuario debe implementar subclases de `VARIABLE` que correspondan a los distintos grupos de variables del problema. En la función `initializeOptimization()` de la clase `MASTER` se construyen todos los objetos correspondientes a las variables involucradas en el problema, que se guardan en el *pool* de variables.

Por ejemplo, supongamos que tenemos un problema cuyas instancias están definidas por un grafo, y cuyo modelo lineal tiene variables x_e (una por cada eje e del grafo) y z_v (una por cada nodo v). Entonces, se pueden definir las clases `VAR_X` y `VAR_Z`, derivadas de `VARIABLE`, que almacenen como datos privados un eje y un nodo, respectivamente. Cada variable x_e será representada como un objeto de clase `VAR_X` con el eje e como dato privado, y sucede lo mismo con las variables z_v del modelo.

Clase `CONSTRAINT`

Las restricciones del problema y los cortes que se agregan durante el proceso se representan como objetos de esta clase. El usuario debe implementar una subclase para cada grupo de restricciones, que se construyen al comienzo del proceso y se guardan en un *pool*. Esta clase tiene el método `coeff(VARIABLE *v)`, que recibe una variable y retorna el coeficiente que acompaña a la variable en la restricción. De esta forma, el entorno puede construir las formulaciones de los subproblemas.

El entorno tiene muchas otras clases, con las cuales el usuario debe interactuar para realizar procesos más complejos. Estas nuevas clases se mencionarán a medida que sean relevantes en la descripción de la implementación realizada.

A.2 Descripción de la implementación

La implementación del algoritmo sigue las convenciones y lineamientos del entorno utilizado, con lo cual es necesario definir subclases de las cuatro clases detalladas en la sección anterior, y redefinir sus métodos, de modo tal de agregar procedimientos específicos.

Datos del problema

En primer lugar, se implementó la clase `PROBLEMA`, que guarda los datos de una instancia particular. Al inicio del proceso de optimización se construye un objeto de esta clase, que contendrá los datos de la instancia que se busca resolver. Este objeto recibe consultas desde todos los puntos del algoritmo, que solamente puede conocer al problema a través de llamadas a este objeto. De esta forma, se "protegen" los datos, que no pueden ser modificados dado que son privados a la clase `PROBLEMA`. Un objeto de esta clase se construye a partir de un

archivo, que contiene los datos del problema[2] (el constructor de la clase recibe el nombre del archivo, y se encarga de inicializar los datos privados del objeto construido). El archivo que contiene el problema debe pasarse como primer argumento en la línea de comandos.

Este archivo debe comenzar con la cantidad de nodos de los grafos, seguida de las listas de adyacencia del grafo de tareas y del grafo de procesadores. Para esto, se indica el grado de cada nodo y el número de cada uno de sus vecinos. Por ejemplo, el siguiente archivo describe la instancia de la figura 2.2 de la página 21:

```
6

3   1 4 2
3   0 3 4
3   0 4 5
3   1 4 5
5   0 1 2 3 5
3   2 3 4

2   1 3
3   0 2 4
2   1 5
2   0 4
3   1 3 5
2   2 4
```

La interface de la clase es sencilla, y permite consultar el grafo G de tareas y el grafo H de procesadores:

`int nodos()`	Cantidad de nodos de los grafos.
`int ejesTareas()`	Cantidad de ejes de G.
`int ejesProcesadores()`	Cantidad de ejes de H.
`int tareasVecinas(i)`	Grado de la tarea i en G.
`int procesadoresVecinos(k)`	Grado del procesador k en H.
`bool sonVecinas(i,j)`	Indica si $ij \in E(G)$.
`bool estanConectados(k,l)`	Indica si $kl \in E(H)$.
`eConj *calcularVecinos(w)`	Conjunto de vecinos de $w \subseteq V(H)$.

El último método recibe un conjunto de procesadores y devuelve un puntero al conjunto de sus vecinos. La clase `ECONJ` implementa conjuntos con elementos enteros entre 0 y n, de modo tal que sus operaciones tienen orden constante (se utiliza un arreglo de bits para representar el conjunto). Esta clase provee las operaciones habituales de conjuntos, incluyendo métodos para agregar y borrar elementos, y para realizar consultas de pertenencia.

Por último, debe mencionarse que los grafos de tareas y procesadores se representan por medio de sus respectivas matrices de adyacencia, lo cual permite realizar consultas en tiempo

[2] En este archivo, las tareas y los procesadores están numerados desde 0 hasta $n-1$, donde n es la cantidad de tareas. Se utiliza esta misma numeración durante el algoritmo.

constante, aunque requiera $O(n^2)$ posiciones de memoria. Se incluyen además dos arreglos que guardan el grado de cada nodo, para que estas consultas se realicen también en tiempo constante.

Variables del modelo

El problema tiene dos grupos de variables, que se implementan como dos subclases de `VARIABLE`. La clase `ASIGNACION` implementa las variables de asignación y_{ik}. Los objetos de esta clase guardan el número i de la tarea y k del procesador, representando así a la variable de asignación. Por su parte, la clase `COSTO` implementa las variables de costo c_{ij}, y los objetos de esta clase están identificados por dos tareas i y j que forman un eje.

Ambas clases tienen funciones que permiten acceder a sus datos. La clase `ASIGNACION` cuenta con las funciones `tarea()` y `procesador()` que devuelven la tarea y el procesador que definen la variable (la variable y_{ik} está definida por la tarea i y el procesador k). Por su parte, la clase `COSTO` contiene las funciones `tarea1()` y `tarea2()`, que proporcionan las dos tareas (extremos de un eje) que definen la variable de costo que el objeto representa. Estas funciones se declaran *inline* por razones de eficiencia, dado que son muy consultadas durante la generación del modelo lineal y la separación de los planos de corte.

Al comienzo del proceso, se construyen los objetos de estas clases, que representan a las variables del modelo. El entorno ABACUS solicita que se declare en este momento el coeficiente que acompaña a cada variable en la función objetivo (es uno de los datos que recibe el constructor de `VARIABLE`), que es nulo para las variables de asignación e igual a 1 para las variables de costo. Estos objetos no se modifican, y se mantienen durante toda la ejecución del algoritmo, dado que éste no incluye técnicas de eliminación y generación de columnas.

Debe observarse que, en realidad, estas dos clases no se derivan directamente de `VARIABLE`, sino que son subclases de `MYVARIABLE`, que es a su vez subclase de `VARIABLE` (ver fig. A.1). La clase `MYVARIABLE` define los métodos `getAsignacion()` y `getCosto()`, que devuelven punteros nulos a `MYVARIABLE`. El primero de estos métodos se redefine en la clase `ASIGNACION` y el segundo en `COSTO`, devolviendo en este caso un puntero al objeto sobre el cual se realiza la llamada. De esta forma, dado un puntero a `MYVARIABLE` se puede decidir de qué clase es el objeto al cual apunta (`ASIGNACION` o `COSTO`): Será una variable de asignación si `getAsignacion()` devuelve un puntero no nulo, y será una variable de costo si sucede lo mismo con `getCosto()`. Todas las funciones que reciben una variable reciben en realidad punteros a `MYVARIABLE`, y se debe decidir de qué tipo es la variable en cuestión, y este mecanismo lo permite[3]

Restricciones y desigualdades

Las restricciones del modelo lineal, las desigualdades válidas y las restricciones que surgen durante el proceso de *branching* se representan como subclases de `CONSTRAINT`. Cada grupo

[3] En compiladores que disponen de *dynamic cast* no es necesario este procedimiento.

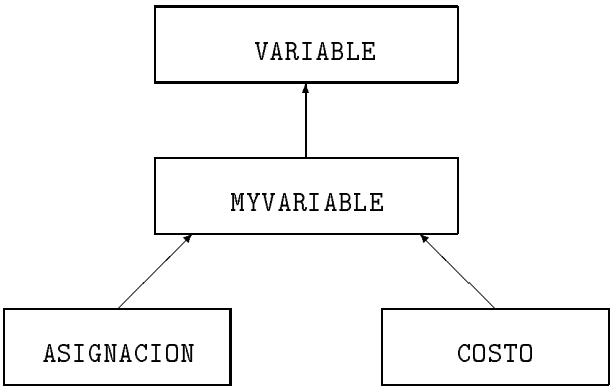


Figura A.1: Estructura de las clases de variables.

de restricciones o familia de desigualdades se implementa como una subclase distinta, que se denomina `CONST`$_n$, con n un número natural. La tabla A.1 describe cada una de estas clases.

`CONST1`	Restricciones de asignación del modelo (tareas).
`CONST2`	Restricciones de asignación del modelo (procesadores).
`CONST3`	Familia 1 de desigualdades.
`CONST4`	Familia 2 de desigualdades.
`CONST5`	Familia 3 de desigualdades.
`CONST6`	Restricciones de costo del segundo modelo.
`CONST7`	Familia 4 de desigualdades.
`CONST8`	Familia 5 de desigualdades.
`CONST9`	Familia 6 de desigualdades.
`CONST10`	Familia 7 de desigualdades.
`CONST11`	Familia 8 de desigualdades.
`CONST13`	Restricciones del *branching* refinado.
`CONST14`	Familia 9 de desigualdades.
`CONST15`	Familia 10 de desigualdades.
`CONST16`	Familia 11 de desigualdades.
`CONST17`	Familia 12 de desigualdades.

Tabla A.1: Clases que implementan restricciones y desigualdades.

Las clases 1, 2 y 6 definen las restricciones del modelo lineal del problema. `CONST1` y `CONST2` representan las restricciones de asignación, y los objetos de estas clases están identificados por una tarea y un procesador, respectivamente. Por último, la clase `CONST6` representa las restricciones de costo, y cada objeto de esta clase se identifica por un eje y un procesador.

Los constructores de estas clases reciben los datos necesarios para identificar sus objetos. Algunas desigualdades están definidas por conjuntos de tareas o procesadores, y sus constructores respectivos reciben como parámetros punteros a `ECONJ`, de los cuales realizan copias privadas que guardan entre sus datos. En el caso de `CONST4` y `CONST15`, se recibe un conjunto de procesadores y es necesario calcular el conjunto de sus vecinos, pero como esto mismo se realiza en los procedimientos de separación, se agregó un constructor que recibe también este conjunto, evitando que se calcule dos veces.

Por otra parte, se definieron los métodos `coeff(MYVARIABLE *v)` de estas clases para que retornen el coeficiente adecuado que acompaña a cada variable. Este método recibe una variable y para decidir cuál es el coeficiente que la acompaña debe decidir su tipo, lo cual se logra mediante funciones adecuadas implementadas en la clase `MYVARIABLE`, del siguiente modo:

```
if( v->getAsignacion() != 0 ) {
        ASIGNACION *y = v->getAsignacion();
        ret = (y->procesador() == elProcesador) ? 1 : 0;
}
else if( v->getCosto() != 0 )
        ret = 0;

return(ret);
```

Este fragmento de código está tomado de la clase `CONST2`, y muestra cómo se utilizan las funciones de las clases `ASIGNACION` y `COSTO` para decidir de qué tipo es la variable `v` que se recibe, y para consultar sus datos.

Por último, debe mencionarse que algunas de las desigualdades requieren del cálculo de coeficientes de *lifting*. En estos casos, los coeficientes se obtienen dentro del constructor (que llama a al método privado `calcularCoeficientes()`) y se guardan en un arreglo para su futura utilización. Para esto, en muchos casos es necesario consultar los datos del problema, y esto se logra pasando al constructor respectivo un puntero al objeto `PROBLEMA` que contiene estos datos.

Clase `MASTER`

Se implementó una sublase de `MASTER`, llamada `MYMASTER` (esta denominación es la sugerida en el manual de ABACUS). El constructor de esta clase recibe el nombre del archivo donde se guardan los datos del problema, y construye un objeto de clase `PROBLEMA`, que permanecerá durante todo el proceso. Además, inicializa los contadores propios del algoritmo (se tienen contadores para cada tipo de cortes, y para las variables fijadas por implicaciones lógicas durante el proceso de *branching*, además de los contadores del entorno), y los parámetros del proceso de optimización.

El método más importante que se redefinió de esta clase fue la función `initializeOptimization()`, encargada de comenzar el proceso de optimización. Esta función inicializa los *pools* de variables y restricciones, y construye las variables y restricciones del problema (un objeto para cada variable y restricción) que se agregan a estos *pools*. En este punto, es necesaria la inicialización del *pool* para los cortes, que comienza con un tamaño de 11 veces la cantidad de restricciones del problema inicial. El siguiente pseudocódigo muestra las acciones que realiza este importante procedimiento:

```
Crear los pools de variables, de restricciones y de cortes;

Agregar al pool de variables los siguientes objetos:
        new ASIGNACION(i,k), para 0 <= i, k < n;
        new COSTO(i,j) para todo eje ij;

Agregar al pool de restricciones los siguientes objetos:
        new CONST1(i) para cada tarea i;
        new CONST2(k) para cada procesador k;
        new CONST6(i,j,k) para cada eje ij y cada procesador k;

Agregar al pool de cortes:
        new CONST5(i) para cada tarea i;
        new CONST14(i) para cada tarea i;

Ejecutar la heuristica primal y guardar su valor como
nueva cota inferior;
```

Las variables y restricciones se guardan primero en buffers especiales (de clase `ABA_BUFFER`), para después inicializar los *pools* con estos buffers. Como se indicó en la sección 6.1, el *pool* de cortes comienza teniendo todas las restricciones de las familias 3 y 9 (clases `CONST5` y `CONST14`, respectivamente). La heurística primal es el método CLO, cuyo resultado se informa al entorno mediante la función `primalBound()`, que recibe el valor de la cota primal.

Esta clase tiene también métodos para acceder al objeto que representa el problema, a los contadores y a las familias de cortes activos. Además, se redefinió la función `terminateOptimization()` para imprimir estadísticas del proceso de optimización cuando finaliza.

Parámetros del algoritmo

Como se mencionó, el archivo `.abacus` contiene los parámetros del algoritmo, cuyos valores se leen al comenzar el proceso. En este archivo, se indican los nombres de los parámetros, seguidos de su valor. Con el objetivo de simplificar el proceso de ajuste de los parámetros, se implementó una forma (soportada por ABACUS) de redefinir estos valores, que consiste en crear un segundo archivo de parámetros, con el mismo formato que el anterior, en el cual figuren los parámetros que se quieren redefinir. Por ejemplo, si queremos utilizar los mismos parámetros que `.abacus`, con excepción de la cantidad de iteraciones por subproblema y el *skip factor*, entonces este archivo debe tener la forma:

```
MaxIterations           3
SkipFactor              4
```

Puede agregarse opcionalmente en la línea de comandos el nombre de este nuevo archivo de parámetros, como segundo argumento. Cuando esto sucede, se lee primero el archivo

`.abacus`, y después se lee el nuevo archivo, sobreescribiendo los parámetros que en él se redefinan. Para poder manejar las dos situaciones, la clase `MYMASTER` tiene dos constructores, el segundo de los cuales recibe un parámetro adicional con el nombre de este nuevo archivo y reinicializa los parámetros, mediante la función `readParameters()` provista por el entorno.

El algoritmo utiliza un segundo archivo de parámetros, llamado `cuts.txt`, que contiene las familias de cortes activos. Por ejemplo, si se quiere que sólo se utilicen las familias 1, 2, 5 y 6 en el método, entonces este archivo debe contener como única línea ``1 2 5 6''. Estos datos son leidos por el constructor de la clase `MYMASTER`, y se guardan en esta misma clase, para ser consultados por los procedimientos de separación. Debe notarse que este mecanismo permite ejecutar distintos grupos de pruebas sin recompilar por completo para cada combinación de familias. Todos los mecanismos relativos a este archivo de familias activas están protegidos por directivas `#ifdef ... #endif`, de modo tal que pueden desactivarse fácilmente borrando la instrucción `#define` correspondiente.

Por último, se implementó la clase `MYSUB`, que es subclase de `SUB`. Se redefinieron muchos métodos de esta clase, para realizar las funciones específicas del proceso, que se describen en la próxima sección.

A.3 Refinamientos al algoritmo

La descripción de la sección anterior corresponde a un algoritmo *branch&bound*, sin agregar cortes ni estrategias específicas para el problema. En esta sección se describen las modificaciones realizadas a dicha implementación para agregar procedimientos de separación y estrategias particulares (fijado de variables por implicaciones lógicas, *branching* especializado, heurísticas, etc.), realizados por medio de modificaciones a la clase `MYSUB`.

A.3.1 Desigualdades válidas y procedimientos de separación

Como se mencionó, cada familia de desigualdades válidas se implementa como una subclase de `CONSTRAINT`, y cada una de ellas tiene los datos que identifican a la desigualdad particular (por ejemplo, la subclase que representa la segunda familia debe tener la tarea y el conjunto de procesadores como datos de la clase), y provee las funciones necesarias para consultarlos.

Los procedimientos de separación se implementan redefiniendo la función `MYSUB::separate()`. Esta función es llamada por ABACUS luego de la resolución de cada subproblema, y se encarga de encontrar cortes entre las desigualdades implementadas. El procedimiento de separación llama a `constraintPoolSeparation()`, que busca las desigualdades del *pool* de cortes que estén violadas en la solución del subproblema, además de llamar en orden a los procedimientos de separación de todas las familias. Estos procedimientos se implementan en los métodos `MYSUB::mySeparate`$_n$`()`, con n de acuerdo al número de la familia que se considera. El siguiente pseudocódigo resume el funcionamiento general de la separación (la variable `newCuts` es un objeto de clase `BUFFER` en el cual se guardan los cortes a medida que se generan).

```
constraintPoolSeparation();
obtenerSolucion();
if( master.estaActivo(1) ) mySeparate1(newCuts);
if( master.estaActivo(2) ) mySeparate2(newCuts);
        ...
if( master.estaActivo(12) ) mySeparate12(newCuts);
nCuts = addCons(newCuts);
return(nCuts);
```

Los procedimientos específicos de separación para cada familia buscan desigualdades violadas en la solución actual, pero para poder hacerlo debe conocerse esta solución. El procedimiento `obtenerSolucion()` lee los valores de las variables en el óptimo del subproblema, y los guarda en las matrices `y` y `c`, de modo tal que, por ejemplo, el valor de la variable y_{ik} se encuentra en `y[i][k]`. Estas matrices son datos privados de la clase `MYSUB`, con lo cual pueden ser consultadas desde los procedimientos de separación, y este acceso es muy eficiente (debe serlo, dado que estos valores se consultan con mucha frecuencia durante la separación).

El procedimiento `mySeparate`$_n$`()` busca desigualdades violadas en la familia n, de acuerdo con las descripciones de la sección 6.1. Cuando se encuentra un corte, debe agregarse al buffer la desigualdad correspondiente. Por ejemplo, consideremos el procedimiento `mySeparate5()`.

```
Para cada eje ij hacer:

    Buscar el procesador k con mayor y[i][k];
    Calcular el lado izquierdo lder de la desigualdad;

    if( c[i][j] > lder && !newCuts.full() ) {
        newCuts.push( new CONST8(master_, this, i, j, k) );
        master->nuevoCorte(8);
    }

Fin (para);
```

La condición `c[i][j] > lder` indica si la desigualdad es violada en la solución actual. En este caso, se encontró un corte que se agrega al buffer `newCuts` de cortes. Los contadores para cada familia (y para los cortes regenerados del *pool*), se mantienen en la clase `MYMASTER` y se actualizan cada vez que se encuentra un nuevo corte, llamando a la función `nuevoCorte()`, que actualiza el contador correspondiente.

Por otra parte, el buffer de cortes para cada iteración[4] se inicializa con espacio para $500\lceil\frac{n}{100}\rceil$ cortes. Es decir, en cada iteración se puede agregar hasta esta cantidad de cortes. Si el espacio se agota, este buffer no es redimensionado en forma automática, con lo cual una vez terminado el espacio, no se pueden agregar más desigualdades. Esta situación se

[4]Se considera una iteración al proceso de resolución de la relajación lineal de un subproblema, junto con sus procedimientos asociados (separación, fijado de variables, heurísticas, etc.)

protege con la condición `!newCuts.full()`, que indica si el buffer tiene espacio disponible. Cuando esto no sucede, los procedimientos de separación no pueden agregar sus cortes al buffer, con lo cual el resultado de su ejecución se pierde. Sin embargo, esto no ocasiona problemas puesto que en muy pocos casos se hallaron más cortes que el máximo permitido en una misma iteración.

Como se mencionó en la sección anterior, el *pool* de cortes se inicializa con espacio para $11r$ cortes, siendo r la cantidad de restricciones de la formulación inicial. Si se agota el espacio, el *pool* es redimensionado en forma automática, aunque esto conlleva una pérdida significativa de performance. En las pruebas que se realizaron, este factor no ocasionó retardos importantes.

A.3.2 Estrategia de *branching*

La función virtual `generateBranchRules()`, de la clase `ABA_SUB` genera las reglas de *branching* necesarias para generar los subproblemas hijos del nodo actual. Para esto, ABACUS provee la clase abstracta `ABA_BRANCHRULE`, de la cual se derivan subclases que representan reglas de *branching* por variables y por restricciones. Esta función devuelve un *pool* de reglas de *branching*, que consituyen el punto de partida para la creación de los subproblemas hijos.

En la implementación original, esta función genera dos hijos, fijando una variable en 1 y en 0 respectivamente (*branching* por variables). Tal como lo prevé el diseño de ABACUS, se modificó la función para implementar la estrategia de *branching* refinado. La nueva función intenta dividir en partes iguales el rango de las variables, buscando que cada subrango contenga variables no nulas. Si esto es posible, se generan dos reglas de *branching* (objetos de clase `ABA_CONBRANCHRULE`, que es subclase de `ABA_BRANCHRULE`), que se utilizarán luego para formar los dos nuevos subproblemas. Si no es posible dividir el rango de ninguna tarea, se llama a la implementación original de la función `generateBranchRules`, para realizar *branching* por variables.

Para llevar a cabo los procesos anteriores, fueron necesarias las siguientes modificaciones al algoritmo:

- Se implementó una nueva subclase de `ABA_CONSTRAINT` que representa las restricciones $\sum_{i=k}^{k'} y_{ik} = 1$. Cuando se generan las dos reglas de *branching*, se asocia un objeto de esta clase a cada una, que representa la restricción asociada con cada subproblema nuevo.
- Se agregaron a la clase `MYSUB` los rangos de las tareas, representados con dos arreglos (índices mínimo y máximo del rango). Para mantener actualizados estos datos, el constructor de esta clase consulta los datos de la regla de *branching* asociada con el subproblema que está en creación.
- Las restricciones asociadas con las reglas de *branching* se ubican en un nuevo *pool*, independiente de los *pools* predefinidos de ABACUS. Este *pool* se agrega como dato privado de la clase `MYMASTER`, y se inicializa con espacio para 5000 restricciones (5000 nodos en el árbol de enumeración), lo cual fue suficiente en todas las pruebas realizadas. De todos modos, si este *pool* agota su espacio, ABACUS lo expande automáticamente, aunque este proceso implica pérdida de performance.

Con esto, se implementa el proceso de *branching* refinado. Es necesario destacar que la inclusión de los rangos de las tareas como dato privado de la clase `MYSUB` simplificó en gran medida la implementación del proceso de fijado de variables por implicaciones lógicas. Por otra parte, el código que se encarga de esta estrategia está protegido por `#ifdef ... #endif`, de modo que puede desactivarse rápidamente el *branching* refinado eliminando la directiva `#define` correspondiente.

A.3.3 Heurística primal

La implementación de la heurística se realiza en un módulo independiente del resto del algoritmo, que provee la siguiente función para ejecutar este método:

```
int clo(problema *prob, solucion &best, int cotaSup);
```

Esta función recibe un puntero al objeto que guarda los datos del problema (primer parámetro) y una cota superior del óptimo (tercer parámetro). Devuelve la mejor solución hallada (el segundo parámetro es de salida) y su valor como resultado.

Una posibilidad es llamar a esta función desde el constructor de la clase `MYMASTER`. De esta forma, se ejecuta la heurística al comienzo del proceso, y su valor resultante se retiene como cota inferior del óptimo. Sin embargo, esta heurística no se ejecuta al comienzo, sino que se espera hasta haber resuelto el primer subproblema (incluyendo la generación de cortes), y la cota superior obtenida con su relajación se informa a la heurística. En cuanto la heurística encuentra una solución con valor igual a la cota superior, se detiene el método dado que se halló el óptimo del problema.

Para implementar este mecanismo, se redefinió la función `improve()` de la clase `ABA_SUB`. Esta función es uno de los puntos de entrada que se ejecutan luego de la optimización de cada subproblema, y está disponible para su redefinición como heurística de mejoramiento primal. Cuando se encuentra en la segunda iteración del primer subproblema (es decir, luego de aplicar cortes al primer nodo), se ejecuta la heurística y se guarda su valor como cota inferior. El siguiente fragmento de código muestra el llamado a la heurística, que tiene particular interés por las funciones del entorno involucradas:

```
int cotaSup = (int) lp()->value();
int vPrimal = clo(prob, cotaSup);

if( master->betterPrimal(vPrimal) ) {
    master->primalBound(vPrimal);
    primalValue = vPrimal;
    return(1);
}
```

La función `lp()` devuelve un puntero a un objeto de clase `ABA_LP` (esta clase es interna al *framework* e implementa los problemas lineales), cuyo método `value()` permite consultar

el valor del óptimo. Cuando se realiza sobre la segunda iteración del primer nodo, este valor es el resultado de la relajación lineal junto con la primer ronda de cortes, y se utiliza como cota superior para el algoritmo heurístico. Si el resultado de la heurística es mejor que la cota inferior actual, se actualiza como nueva cota inferior, por medio de la función `primalBound()` de la clase `MASTER`.

A.3.4 Heurística de mejoramiento primal

El procedimiento descripto se implementó redefiniendo la función `improve()` de la clase `ABA_SUB`, para que, además de ejecutar la heurística primal en el primer subproblema, se ejecute la heurística de mejoramiento en los restantes nodos. De acuerdo con las convenciones de ABACUS, si se encuentra un mejor valor que la cota inferior global, se actualiza el parámetro `primalBound` y se devuelve `1`, retornando `0` en caso contrario.

A.3.5 Fijado de variables por implicaciones lógicas

La implementación de este paso se realizó redefiniendo la función

```
void ABA_SUB::setByLogImp( ABA_BUFFER<int> &variables,
                           ABA_BUFFER<ABA_FSVARSTAT*> &status );
```

En la implementación original, esta función no realiza ninguna acción, y está disponible para su redefinición. Se realiza este procedimiento cuando se utiliza la estrategia de *branching* refinado, en cuyo caso se tienen para cada subproblema dos arreglos `minProc` y `maxProc`, que indican para cada tarea el rango de procesadores sobre el cual se puede asignar. Para cada eje ij del grafo de tareas, se consideran los rangos de las tareas i y j y se verifica si existe algún eje entre estos rangos de procesadores. Si esto no sucede, se fija la variable c_{ij} en cero.

La función `setByLogImp()` recibe dos parámetros por referencia. En el primero de ellos deben guardarse los índices de las variables que se fijan, y en el segundo se guardan (en el mismo orden) los valores en los cuales estas variables deben fijarse. El siguiente pseudocódigo resume esta descripción.

```
Para cada eje ij del grafo de tareas hacer:

    Buscar los rangos de las tareas i y j;
    Decidir si existe un eje entre estos rangos;

    Si la variable c[i][j] debe fijarse en cero:
        Buscar el indice s de la variable c[i][j];
        variables.push(s);
        status.push( new ABA_FSVARSTAT( setToLowerBound ) );
    Fin
```

```
Fin (para)
```

Dado un índice `s`, la función `variable(s)` de la clase `MASTER` retorna un puntero al objeto que representa esta variable.

Bibliografía

[Aar/95a] Aardal K. y van Hoesel S., *Polyhedral Techniques in Combinatorial Optimization I: Theory*. Utrecht University.

[Aar/95b] Aardal K. y van Hoesel S., *Polyhedral Techniques in Combinatorial Optimization II: Computations*. Utrecht University.

[Asc/97] Ascheuer N., Jünger M. y Reinelt G., *A Branch & Cut Algorithm for the Asymmetric Hamiltonian Path Problem with Precedence Constraints*. Konrad-Zuse-Zentrum für Informationstechnik, Berlin. Preprint SC 97-70 (1997).

[Bal/93] Balcázar J., *Programación Metódica*. McGraw-Hill (1993).

[Bar/98] Baruah S., *The Multiprocessor Scheduling of Precedence-Constrained Task Systems in the Presence of Interprocessor Communication Delays*. Operations Research, Vol. 46 No. 1 (1998).

[Ber/85] Berge C., *Graphs*. North-Holland (1985).

[Bok/81] Bokhari S., *On the Mapping Problem*. IEEE Transactions on Computers, Vol. C-30 No. 3 (1981).

[Bur/91] Burkard R., Karisch S. y Rendl F., *QAPLIB - A Quadratic Assignment Problem Library*. European Journal of Operational Research, 55 (1991).

[Bre/75] Brearley A., Mitra G. y Williams H., *Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm*. Mathematical Programming, Vol. 8 (1975).

[Cho/95] Chockalingam T. y Arunkumar S., *Genetic Algorithm Based Heuristics for the Mapping Problem*. Computers Ops. Res., Vol. 22 No. 1 (1995).

[Chr/98] Christof T. y Reinelt G., *Decomposition and Parallelization Techniques for Enumerating the Facets of 0/1-Polytopes*. Institut für Angewandte Mathematik, Universität Heidelberg (1998).

[Col/95] Coll P., Durán G. y Robak D., *Un GRASP para el problema de coloreo de grafos*. Anales de las II Jornadas Uruguayas de Investigación Operativa, Montevideo, Uruguay (1995).

[Cow/95] Cowen L., Feigenbaum J. y Kannan S., *A Formal Framework for Evaluating Heuristic Programs*. DIMACS Technical Report 95-27 (1995).

[Cpl/94] CPLEX Optimization Inc., *Using the CPLEX Callable Library, including Using the CPLEX Linear Optimizer with CPLEX Barrier and Mixed Integer Solvers.* Version 3.0 (1994).

[Erc/90] Ercal F., Ramanujam J. y Sadayappan P., *Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning.* Journal of Parallel and Distributec Computing, 10 (1990).

[Feo/95] Feo T. y Resende M., *Greedy Randomized Adaptive Search Procedures.* Journal of Global Optimization (1995).

[Fer/93] Ferreira C., Martin A. y Weismantel R., *A Cutting Plane Based Algorithm for the Multiple Knapsack Problem.* Konrad-Zuse-Zentrum für Informationstechnik, Berlin. Preprint SC 93-7 (1993).

[For/96] Fortin S., *The Graph Isomorphism Problem.* University of Alberta, Technical Report TR 96-20 (1996).

[Gar/75] Garey M. y Johnson D., *Complexity Results for Multiprocessor Scheduling under Resource Contraints.* SIAM J. Comput. 4 (1975).

[Gar/79] Garey M. y Johnson D., *Computers and intractability: A guide to the theory of NP-completeness.* W. H. Freeman and Company (1979).

[Glo/95] Glover F., *Tabu Search Fundamentals and Uses.* University of Colorado, Technical Report (1995).

[Gro/93a] Grötschel M. y Lovász L., *Combinatorial Optimization.* Konrad-Zuse-Zentrum für Informationstechnik, Berlin. Preprint SC 93-13 (1993).

[Gro/88] Grötschel M., Lovász L. y Schrijver A., *Geometric Algorithms and Combinatorial Optimization.* Springer-Verlag (19889.

[Gro/92] Grötschel M., Martin A. y Weismantel R., *Routing in Grid Graphs by Cutting Planes.* Konrad-Zuse-Zentrum für Informationstechnik, Berlin. Preprint SC 92-26 (1992).

[Gro/93b] Grötschel M., Monma C. L. y Stoer M., *Design of Survivable Networks.* Konrad-Zuse-Zentrum für Informationstechnik, Berlin. Preprint SC 93-9 (1993).

[Gui/81] Guignard M. y Spielberg K., *Logical Reduction Methods in Zero-One Programming.* Opeations Research, Vol. 29 No. 1 (1981).

[Hah/98] Hahn P. y Grant T., *Lower Bounds for the Quadratic Assignment Problem Based Upon a Dual Formulation.* Operations Research, Vol. 46 No. 6 (1988).

[Har/68] Harary F., *Graph Theory.* Addison-Wesley Publishing Company (1968).

[Hil/95] Hillier F. y Lieberman G., *Introduction to Operations Research.* Sexta edición, McGraw-Hill International Editions (1995).

[Hoe/95] Hoese A., *Nuevo Algoritmo de Optimización Combinatorial Aplicado a la Evaluación de un Sistema Híbrido Aislado de Generación de Energía Eléctrica de Baja Potencia.* XXIV Jornadas Argentinas de Informática e Investigación Operativa (1995).

[Joh/98] Johnson E., Nemhauser G. y Savelsbergh M., *Progress in Linear Programming Based Branch-and-Bound Algorithms: An Exposition.* Technical Report, Georgia Institute of Technology (1998).

[Jun/96] Jünger M. y Kaibel V., *A Basic Study of the QAP-Polytope.* Angewandte Mathematik und Informatik, Universität zu Köln. Report No. 96.215 (1996).

[Jun/97a] Jünger M. y Kaibel V., *The QAP-Politope and the Star-Transformation.* Universität zu Köln, Technical Report (1997).

[Jun/97b] Jünger M. y Kaibel V., *Box-Inequalities for Quadratic Assignment Polytopes.* Universität zu Köln, Technical Report (1997).

[Jun/92] Jünger M., Reinelt G. y Thienel S., *Provably good solutions for the traveling salesman problem.* Universität zu Köln, Technical Report 92.114 (1992).

[Jun/94] Jünger M., Reinelt G. y Thienel S., *Practical Problem Solving with Cutting Plane Algorithms in Combinatorial Optimization.* Interdisziplinares Zentrum für Wissenschaftliches Rechnen, Universität Heidelberg. Preprint 94-24 (1994).

[Jun/97c] Jünger M. y Thienel S., *The Design of the Branch and Cut System ABACUS.* Universität zu Köln, Technical Report 97.260 (1997).

[Jun/97d] Jünger M. y Thienel S., *Introduction to ABACUS - A Branch and Cut System.* Universität zu Köln, Technical Report 97.263 (1997).

[Lee/87] Lee S. y Aggarwal J., *A Mapping Strategy for Parallel Processing.* IEEE Trans. Computers, Vol. C-36 No. 4 (1987).

[Lov/72] Lovász L., *Normal Hypergraphs and the Perfect Graph Conjecture.* Discrete Math, Vol. 2 (1972).

[Mac/95] Maculan N., Porto S., Ribeiro C. y de Souza C., *A New Formulation for Scheduling Unrelated Processors under Precedence Constraints.* Pontifícia Universidade Católica do Rio de Janeiro (1995).

[Mag/97] Magalhães E. y de Souza C., *The Edge-Weighted Clique Problem: valid inequalities, facets and polyhedral computations.* UNICAMP, Reporte Técnico IC-97-14 (1997).

[Mag/86] Magnanti T., Mireault P. y Wong R., *Tailoring Benders Decomposition for Uncapacitated Network Design.* Mathematical Programming Study, Vol. 26 (1986).

[Mag/81] Magnanti T. y Wong R., *Accelerating Benders Decomposition: Algorithmic Enhancement and Model Selection Criteria.* Operations Research, Vol. 29 No. 3 (1981).

[Mar/96] Martin O. y Otto S., *Combining simulated annealing with local search heuristics.* Annals of Operations Research, Vol. 63 (1996).

[Mon/95] Monakhov O., *Parallel Algorithm for Mapping of Parallel Programs into Pyramidal Multiprocessor.* Lecture Notes in Computer Science 1041 (1995).

[Nem/88] Nemhauser G. y Wolsey L., *Integer Programming and Combinatorial Optimization.* John Wiley & Sons (1988).

[Pel/99] Pellegrini F., *SCOTCH and LIBSCOTCH 3.3 user's guide.* Université Bordeaux I (1999).

[Por/93] Porto S. y Ribeiro C., *A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints.* Pontifícia Universidade Católica do Rio de Janeiro, MCC03/93 (1993).

[Pra/95] Prawda J., *Métodos y Modelos de Investigación de Operaciones. Volumen I.* Noriega Editores (1995).

[Res/95] Resende M., Ramakrishnan K. y Drezner Z., *Computing Lower Bounds for the Quadratic Assignment Problem with an Interior Point Algorithm for Linear Programming.* Technical Report AT&T Laboratories.

[Sad/87] Sadayappan P. y Ercal F., *Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes.* IEEE Transactions on Computers, Vol. C-36 No. 12 (1987).

[Thi/95] Thienel S., *ABACUS - A Branch and Cut System.* Tesis Doctoral, Universität zu Köln (1995).

[Thi/96] Thienel S., *A Simple TSP-Solver: An ABACUS Tutorial* (1996).