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

Tesis de Licenciatura

Métodos basados en programación lineal entera para el problema de planificación de corrugadoras por tramos consecutivos

Alumno: Silvina Lucero
LU: 641/02
svlucero@gmail.com

Director: Dr. Javier Marenco
jmarenco@dc.uba.ar

CoDirector: Lic. Federico Martinez
femartinez@dc.uba.ar

Agradecimientos

A Javi por su infinita paciencia, sus consejos, sus explicaciones y sus correcciones. A Fede por las buenas ideas, la buena onda y la completa predisposición. A Leo por aguantarme durante este período, por prestarme a Josephina para los experimentos, por darme aliento y por estar siempre incondicional a mi lado. A mi familia y amigos por escucharme y apoyarme en todo momento.

Resumen

Una corrugadora es una máquina capaz de realizar cortes longitudinales y transversales para formar planchas de cartón de diferentes dimensiones. El *problema de cortes por tramos consecutivos* (PCTC) consiste en encontrar un programa de cortes que cumpla un conjunto de pedidos minimizando el desperdicio de materia prima, con la restricción extra de que no se puede comenzar la producción de un pedido si hay más de un pedido con producción parcial y aún no terminado.

El PCTC surge en el contexto de máquinas corrugadoras que no tienen mucho espacio físico a la salida de la máquina, con lo cual no es posible tener muchos bancales abiertos en un momento dado (en cada momento de la producción se tiene un bancal abierto por cada pedido en producción y aún no terminado). Este problema está relacionado con el *problema de minimización de discontinuidades* (MDP), que consiste en encontrar un programa de cortes que minimice la cantidad de veces que un pedido se recomienza dentro de dicho programa [6, 10]. En nuestro caso particular, una solución óptima para el MDP con cero discontinuidades –si existe– corresponde a una solución factible para el PCTC.

En esta tesis se propone un enfoque basado en programación lineal entera para el PCTC. Se plantean cuatro modelos de programación lineal entera y se realizan experimentos computacionales con el objetivo de determinar cuál de ellos tiene mejor performance en la práctica. Se presentan familias de desigualdades válidas para uno de estos modelos, se diseñan procedimientos de separación para las familias de desigualdades halladas y se propone un algoritmo *branch and cut*. Finalmente, se propone una heurística primal basada en programación entera, y se evalúa su impacto computacional sobre instancias reales.

Abstract

A corrugator is a machine performing longitudinal and transversal cuts in order to produce rectangular pieces of corrugated cardboard. The *consecutive combinations cutting problem* (CCCP) consists in finding a cutting program satisfying a set of orders while minimizing the total waste, with the additional constraints that production of an order cannot start if there exists more than one unfinished order with partial production.

The CCCP arises in the context of corrugators with small space at the machine outlet, hence it is not possible to have many open pallets at any moment in time (there exists an open pallet for each unfinished order). This problem is closely related to the *minimization of discontinuities problem* (MDP), which consists in finding a cutting program minimizing the number of times that production of an unfinished order is restarted [6, 10]. In our particular case, an optimal solution for the MDP –if exists– corresponds to a feasible solution to the CCCP.

In this thesis we propose an integer programming approach for the CCCP. Four integer programming models are presented, and computational experiments are performed in order to determine which of them has the best performance. Several families of valid inequalities are presented, together with associated separation procedures. Finally, an integer-programming-based primal heuristic is proposed, and its computational impact is evaluated on real-life instances.

Índice general

1 Introducción **3**
- 1.1 Programación lineal entera . . . 3
- 1.2 Cartón corrugado . . . 4
- 1.3 Problema de cutting stock . . . 5
- 1.4 Problema de cortes por tramos consecutivos . . . 6
- 1.5 Resumen del trabajo . . . 8

2 Modelos de programación lineal entera **9**
- 2.1 Modelo 1: Anillo-Estrella . . . 9
- 2.2 Modelo 2: Grafo caterpillar . . . 12
- 2.3 Modelo 3: Caterpillar con cortes . . . 14
- 2.4 Modelo 4: Cuadrático . . . 16
- 2.5 Comparaciones . . . 17
  - 2.5.1 Análisis de los tiempos de ejecución . . . 17
  - 2.5.2 Características de las soluciones obtenidas . . . 20

3 Desigualdades válidas **24**
- 3.1 Familias de desigualdades válidas . . . 24
  - 3.1.1 Desigualdades Garra . . . 24
  - 3.1.2 Desigualdades Ciclo . . . 25
  - 3.1.3 Desigualdades Vecinos . . . 26
  - 3.1.4 Desigualdades para los nodos internos y externos . . . 26
- 3.2 Algoritmos de separación . . . 26
- 3.3 Experiencia computacional . . . 27
  - 3.3.1 Desigualdades Garra . . . 30
  - 3.3.2 Desigualdades Ciclo . . . 30
  - 3.3.3 Desigualdades Vecinos . . . 30
  - 3.3.4 Desigualdades para los nodos internos y externos . . . 30
- 3.4 Conclusiones . . . 34

4 Heurística primal **39**
- 4.1 Desarrollo . . . 39
  - 4.1.1 Ejemplo . . . 40
- 4.2 Experiencia computacional . . . 43

5 Conclusiones **48**

A Implementación **50**
A.1 Generación y ejecución de los modelos 50
A.1.1 Archivos auxiliares 50
A.1.2 Algoritmos asociados 51
A.2 Algoritmos Generales 53
A.2.1 Armado de un grafo a partir de una solución 53
A.2.2 Otros algoritmos 53
A.3 Algoritmo branch and cut 56
A.3.1 LazyConstraintCallback: Ciclo y Garra 56
A.3.2 IncumbentCallback: Validación del grafo 58
A.3.3 UserCutCallback: Cortes 58
A.3.4 HeuristicCallback 62

B Programa PCTC **64**

Capítulo 1

Introducción

1.1 Programación lineal entera

Los orígenes de la disciplina conocida como *investigación operativa* se remontan a la segunda guerra mundial. Durante la guerra, el gobierno británico convocó a prestigiosos científicos para ayudar a encontrar buenas estrategias en la planificación de recursos a fin de reducir los costos del ejército y contribuir a los objetivos bélicos. En 1942, George Dantzig aplicó por primera vez una versión preliminar del *método simplex* para determinar la localización óptima de las antenas y la mejor distribución de sus señales. Gracias a este estudio, el ejército británico consiguió duplicar la efectividad del sistema de defensa y evitar ataques aéreos alemanes. El método quedó en secreto hasta 1947 cuando Dantzig lo formuló en términos matemáticos. Ese mismo año John von Neumann desarrollo la teoría de dualidad para programación lineal [15].

Al finalizar la segunda guerra mundial, se observó que estas herramientas desarrolladas en un contexto bélico también podían ser aplicadas para resolver problemas no militares. Con estos fines, en 1947 se crea un grupo llamado *scientific computation of optimum programs* (SCOOP). Muchos científicos importantes de la época fueron convocados. Habitualmente se considera que con la creación de este grupo se formalizó la existencia de la investigación operativa [12].

En 1948, durante la guerra fría, la URSS bloqueó las comunicaciones terrestres con Berlín siendo imposible para Estados Unidos e Inglaterra proporcionar suministros para dicha ciudad. El grupo SCOOP liderado por Dantzig diseñó un plan de abastecimiento aéreo que, en poco tiempo, logró igualar los suministros realizados por carretera y ferrocarril [16].

Desde entonces, el uso de modelos de programación lineal y programación lineal entera para resolver problemas de optimización combinatoria ha ido en aumento, ayudado por el incremento de la potencia de cómputo y la mejora en los algoritmos y paquetes de software que los implementan. Este tipo de técnicas es usado en diversas áreas como telecomunicaciones, planificación de tareas en entornos industriales, problemas de logística, etc.

Un problema de *programación lineal* es el problema de optimizar una función objetivo lineal sobre un conjunto de puntos definido por desigualdades lineales en las variables. Un problema de *programación lineal entera* es un problema lineal que requiere que todas sus variables sean enteras, si dichas variables sólo pueden tomar valores en $\{0, 1\}$ entonces se dice que es un *problema de programación entera 0-1*. Si algunas de las variables pueden tomar valores enteros y otras pueden tener valores fraccionarios se dice que es un *problema*

de programación lineal mixto.

Cuando el problema de programación lineal es entero o mixto, se suelen utilizar algoritmos basados en programación lineal con un esquema denominado *branch and bound*. Este algoritmo utiliza la técnica de *divide and conquer* dividiendo recursivamente el problema original en subproblemas del mismo tipo pero idealmente más sencillos. Se genera así un *árbol de enumeración* cuyos nodos son los subproblemas. En cada nodo se resuelve la *relajación lineal* del subproblema (es decir, el problema de programación lineal obtenido eliminando las restricciones de integralidad sobre las variables) y el resultado permite obtener una cota dual del óptimo del subproblema. La adición de *planos de corte* permite además ajustar la relajación lineal agregando restricciones lineales válidas. Existen esquemas generales de generación de planos de corte (como por ejemplo los *cortes de Gomory*), y es habitual también estudiar familias de planos de corte para problemas particulares. Si durante el procedimiento *branch and bound* se aplican planos de cortes, se dice que es un algoritmo *branch and cut*. Este tipo de algoritmos resulta útil para muchos problemas si se encuentran familias de desigualdades válidas que ajusten significativamente la relajación lineal de la formulación inicial [17].

1.2 Cartón corrugado

Las cajas de cartón corrugado se utilizan para transporte de mercadería, desde frutas y verduras hasta máquinas industriales. La función primordial es proteger los elementos que contienen mediante un sistema especial de amortiguación. Sus características livianas, resistentes y económicas las han convertido en el método más popular y efectivo de embalaje para trasportar bienes manufacturados.

El cartón corrugado (ver Figura 1.1) está formado por capas de papel, habitualmente llamadas *liners* y *ondas*. Los liners son papeles sin ondular, mientras que los ondas son papeles acanalados que se adhieren a los liners mediante pegamento. Existen varios tipos de cartón corrugado: corrugado de una cara (un liner y un onda), corrugado sencillo (dos liners y un onda), doble corrugado (tres liners y dos ondas intercalados) y triple corrugado (cinco liners y dos ondas intercalados).

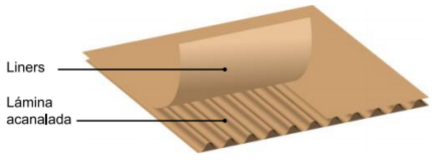

Figura 1.1: Papel corrugado. [3]

Una línea de producción de cartón corrugado es una secuencia de pasos donde se trasforman bobinas de papel en cajas de cartón. El papel es tratado mediante diferentes niveles de calor y humedad para hacerlo más flexible al momento del corrugado, luego se realiza el proceso de acanalado hasta obtener la onda. La siguiente fase consiste en pegar los liners y las ondas, para lo cual se utiliza calor y adhesivo. Por último, este flujo continuo de cartón corrugado pasa por una etapa de cortes longitudinales y transversales para formar *planchas* rectangulares según las especificaciones puntuales de cada pedido.

Estas planchas se estacionan en bancales o pallets durante unas horas, y luego pasan por los procesos de impresión, troquelado y pegado. Por ejemplo, la Figura 1.2 muestra una

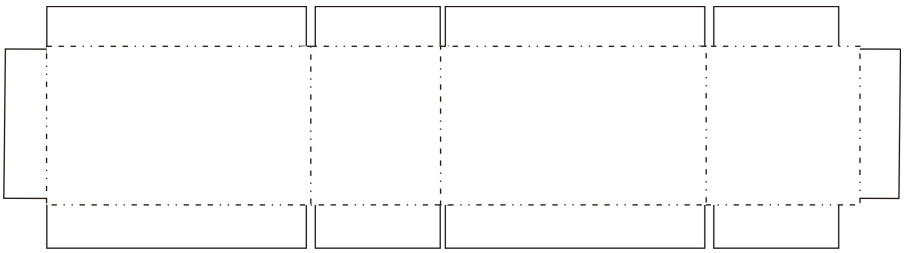

Figura 1.2: Plancha de cartón

plancha troquelada. Las líneas punteadas corresponden a los *trazadores*, que se utilizan para plegar la caja y hacer que tome su forma final. El resultado final es una caja impresa y pegada.

1.3 Problema de cutting stock

El *problema de cutting stock* (*CSP*) es un problema muy estudiado por sus aplicaciones industriales (papel, vidrio, cerámica, etc.). En un sentido amplio, el problema consiste en encontrar un plan de cortes para pedidos de diferentes configuraciones de modo tal de minimizar el desperdicio de la materia prima usada. En este trabajo nos enfocaremos en el estudio del *CSP* con una dimensión [14] para el armado de cajas de cartón. En este contexto, se busca minimizar el desperdicio de cartón.

La primer formulación del *CSP* fue dada en el año 1939 por Leonid Kantorovich [8]. En 1961–1963, Paul Gilmore y Ralph Gomory lograron avances importantes en la resolución del problema para una dimensión al proponer una estrategia eficiente que utiliza *patrones de corte* [4, 5].

Un patrón está dado por una lista de pedidos que es factible de cortar en forma transversal en la corrugadora. Sea Q el conjunto de patrones. Para cada $j \in Q$, se define la variable x_j como el número de veces que se usa el patrón j. Sea además P el conjunto de pedidos. Para $i \in P$ y $j \in Q$, el valor a_{ij} representa la cantidad de veces que aparece el pedido i en el patrón j y el valor c_j representa el costo de utilizar el patrón j. Finalmente, para cada $i \in P$ llamamos q_i a la cantidad de artículos (planchas, en nuestro caso) solicitados por el pedido i. Con estas definiciones, el CSP se define del siguiente modo:

$$\begin{aligned} \text{mín} \quad & \sum_{j \in Q} c_j \, x_j & & \\ \sum_{j \in Q} a_{ij} \, x_j \quad & \geq \quad q_i & \forall \, i \in P & \qquad (1.1) \\ x_j \quad & \in \quad \mathbb{Z}_+ & \forall \, j \in Q & \qquad (1.2) \end{aligned}$$

La restricción (1.1) asegura que la cantidad producida sea, como mínimo, la cantidad solicitados q_i para la instancia i y la restricción (1.2) establece que la cantidad de patrones usados para el patron i sea entera y no negativa.

La cantidad de patrones crece exponencialmente con la cantidad de pedidos, lo que hace imposible enumerar las variables. En 1961 Gilmore y Gomory aplicaron por primera vez una técnica que permite manejar casos en los cuales la cantidad de variables es grande pero con una cantidad de restricciones relativamente pequeña. Esta técnica recibió el nombre de

generación de columnas. Dicho algoritmo comienza con un problema maestro utilizando un patrón trivial e itera buscando patrones que mejoren la función objetivo. En cada paso, se utiliza una instancia especial del problema de la mochila para determinar si existe un patrón que mejora la función objetivo [4, 5].

1.4 Problema de cortes por tramos consecutivos

Definimos en esta sección el *problema de cortes por tramos consecutivos* (PCTC) sobre el que trabajamos en esta tesis. En el contexto del PCTC, un *pedido* está especificado por una terna $P_i = (a_i, l_i, q_i) \in \mathbb{R}_+^2 \times \mathbb{Z}_+$ compuesta por el ancho a_i y el largo l_i de las planchas, junto con la cantidad q_i de planchas a fabricar [11]. Típicamente, una corrugadora cuenta con la posibilidad de fabricar al mismo tiempo más de un pedido, dado que incluye más de una mesa de corte transversal además de un conjunto de cuchillas longitudinales. La configuración habitual incluye dos mesas, con la posibilidad de combinar entre sí pedidos de hasta dos largos distintos [2], y asumimos en este trabajo que ésta es la situación.

Llamamos $A \in \mathbb{R}_+$ al ancho de la corrugadora y $\gamma \in [0, 1]$ al porcentaje de *refile máximo*. Suponemos dado un conjunto de pedidos $P = \{1, \ldots, n\}$, de modo tal que cada pedido i tiene asociada una terna P_i como en el párrafo anterior. Una *combinación* es una tupla $C = (i, j, n_1, n_2, m) \in P^2 \times \mathbb{Z}_+^2 \times \mathbb{R}_+$, donde n_1 y n_2 representan la cantidad de cortes longitudinales para el pedido i y j respectivamente, de modo tal que $(1 - \gamma)A \leq n_1 a_i + n_2 a_j \leq A$ (es decir, es una secuencia de planchas de dos pedidos cuyos anchos sumados no exceden el ancho de la corrugadora ni están por debajo de la tolerancia para el refile). El valor m representa los *metros lineales* que se fabrican de la combinación. La cantidad de planchas programadas del pedido i en la combinación es $\lfloor m\, n_1/l_i \rfloor$, y la cantidad de planchas programadas del pedido j en la misma es $\lfloor m n_2/l_j \rfloor$. En el contexto de la industria del cartón corrugado, una combinación a veces se denomina un *tramo*.

Un *programa de cortes* es una secuencia $\mathcal{S} = (C_1, \ldots, C_k)$ de combinaciones, de modo tal que para cada pedido $i \in P$, la suma de las cantidades programadas del pedido i en todas las combinaciones está dentro del intervalo $[q_i(1-p), q_i(1+p)]$, donde $p \in [0, 1]$ es un porcentaje de tolerancia (típicamente, $p \leq 0{,}05$). Un problema clásico en este contexto es el problema de *cutting stock* bidimensional que, dado un conjunto de pedidos P, un ancho de corrugadora A y un porcentaje de refile máximo γ, pide encontrar un programa de cortes factible –si existe– que minimice el refile total.

Un *programa de cortes por tramos consecutivos* es un programa de cortes $\mathcal{S}$ tal que para cada pedido $i \in P$, las combinaciones en las que aparece el pedido i aparecen en forma consecutiva en $\mathcal{S}$. Dado un conjunto de pedidos P, un ancho de corrugadora A y un porcentaje de refile máximo γ, el problema de *programación de corrugadoras por tramos consecutivos* (PCTC) consiste en hallar un programa de cortes por tramos consecutivos. Adicionalmente, se puede solicitar que el programa obtenido sea el que minimiza el refile total entre todos los programas por tramos consecutivos.

Por ejemplo, la Tabla 1.1 muestra una instancia del PCTC, compuesta por un conjunto de pedidos. Para cada pedido, se especifica el ancho y largo de la plancha y la cantidad de planchas a fabricar. Para dicha instancia, la Figura 1.3 muestra una solución factible al problema PCTC. Cada fila corresponde a una combinación, y a la derecha se muestran los metros lineales correspondientes a cada una. Esta solución está compuesta por cinco combinaciones. Las dos primeras consisten de los pedidos 4 y 5 “autocombinados”, respectivamente. Luego

Pedido	Ancho	Largo	Cantidad
pedido 1	476	476	950
pedido 2	455	455	2000
pedido 3	450	450	2000
pedido 4	450	450	2000
pedido 5	405	405	3120

Cuadro 1.1: Instancia de ejemplo

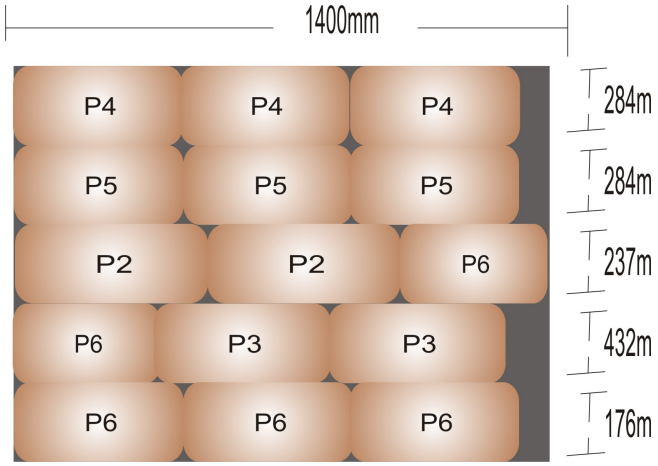

Figura 1.3: Solucion PCTC para la instancia 3

se fabrican los pedidos 2 y 3 junto con el pedido 6, que finaliza la secuencia autocombinado. Si la última combinación se intercambia con la primera, entonces la solución ya no es factible para el PCTC porque el pedido 6 no se estaría fabricando en combinaciones consecutivas. Es interesante notar que no siempre se pueden reordenar las combinaciones de una programación que cumple las cantidades de todos los pedidos (pero que no es una solución por tramos consecutivos) para que sea una solución factible del PCTC.

El PCTC está relacionado con el *problema de minimización de discontinuidades* (MDP), que consiste en encontrar un programa de cortes que minimice la cantidad de veces que un pedido se recomienza dentro del programa de cortes [6, 10]. En nuestro caso particular, una solución óptima para el MDP con cero discontinuidades –si existe– corresponde a una solución factible para el PCTC. Otros problemas relacionados son el *problema de minimización de bancales abiertos* (*minimization of open stack problem*, MOSP) que pide minimizar el número máximo de bancales abiertos en todo momento, y el *problema de minimización de la dispersión de órdenes* (*minimization of order spread problem*, MORP) que pide minimizar el tiempo máximo que un bancal permanece abierto. El MOSP ha sido muy estudiado en la literatura especializada [6], pero tanto el MORP como el MDP no han recibido tanta atención. No estamos al tanto de estudios sobre el MDP con cero discontinuidades aplicados al problema de minimización del refile en corrugadoras. Por otro lado, el PCTC es estudiado en [13] donde se presentan dos heurísticas basadas en grafos *caterpillar*. En dicho trabajo se plantea un modelo basado en patrones y se utiliza la estructura de los grafos *caterpillar* para llegar a una solución factible.

1.5 Resumen del trabajo

Este trabajo tiene por objetivo estudiar el problema PCTC por medio de técnicas de programación lineal entera. Se plantean cuatro modelos diferentes que surgen de distintos enfoques, incluyendo (a) adaptaciones de modelos planteados previamente para el problema de *m-anillo-estrella*, (b) la observación de que toda solución del PCTC puede ser modelada utilizando *grafos caterpillar* y (c) la enumeración de patrones de pedidos válidos. De estos modelos implementados se selecciona aquel cuyo tiempo de optimización es mejor utilizando una muestra reducida de 34 instancias de pedidos reales con diferentes configuraciones. Sobre dicho modelo se estudian familias de desigualdades válidas y se propone un algoritmo de tipo *branch and cut*. Después de analizar los resultados de este algoritmo, se propone y estudia una heurística primal basada en programación lineal entera.

El trabajo continua organizado de la siguiente manera:

- Capítulo 2 - *Modelos de programación lineal entera*: Este capítulo detalla cuatro modelos de programación lineal entera planteados para el PCTC, se explican los conceptos básicos de cada idea de modelado y se hace una comparación empírica de estos modelos. Finalmente, se estudian las características de las soluciones arrojadas por cada modelo.

- Capítulo 3 - *Desigualdades válidas*: De los cuatros modelos se selecciona aquel para el cual un algoritmo *branch and bound* obtuvo los mejores resultados computacionales. Este capítulo estudia diferentes familias de desigualdades para dicho modelo, junto con algoritmos de separación para estas familias. Se presentan resultados computacionales que muestran que las desigualdades halladas no proporcionan mejoras importantes, y se ofrece una hipótesis que podría explicar este hecho.

- Capítulo 4 - *Heurística primal*: Se propone e implementa una heurística primal basada en programación lineal entera para el problema, con la intención de mejorar las cotas primales durante la ejecución de un algoritmo *branch and bound*. Se analizan distintos criterios de parada y se presentan resultados computacionales.

- Capítulo 5 - *Conclusiones*: Se mencionan las conclusiones del trabajo y algunas líneas para continuar a futuro.

Capítulo 2

Modelos de programación lineal entera

Presentamos en este capítulo cuatro modelos alternativos de programación lineal entera para el PCTC. Se realizan a continuación experimentos computacionales sobre instancias reales, para determinar cuál de ellos tiene mejor performance, con el objetivo de tomarlo como base para el resto del trabajo.

Llamamos $P = \{1, \ldots, n\}$ al conjunto de pedidos y cada pedido $j \in P$ tiene asociada una cantidad q_j a fabricar. Sea $\gamma_{ij} \in \mathbb{Z}_+$ la cantidad de combinaciones en las que participan los pedidos i y j, y denotamos por n^i_{ijc} a la cantidad de apariciones del pedido i en la c-ésima combinación entre los pedidos i y j, para $c = 1, \ldots, \gamma_{ij}$. Si un pedido $i \in P$ se *autocombinar*, entonces tenemos $\gamma_{ii} > 0$.

Sea $G = (P, E)$ el *grafo de combinaciones*, que tiene un nodo por cada pedido y una arista ij por cada par de pedidos i, j tales que $\gamma_{ij} > 0$. Es importante observar que el grafo de combinaciones es un *pseudografo* si existen pedidos autocombinados.

Llamamos a_i y l_i al ancho y largo de las cajas del pedido i, respectivamente. Para $i, j \in P$ y $c = 1, \ldots, \gamma_{ij}$, definimos $MAX_{ijc} = \text{máx}(l_i\, q_i/n^i_{ijc}, l_j\, q_j/n^j_{ijc})$ como una cota superior a los metros lineales que se fabrican en la c-ésima combinación entre los pedidos i y j. Además, definimos el *refile* $D_{ijc} = A - (n^i_{ijc}\, a_i + n^j_{ijc}\, a_j)$ en centímetros de esta combinación, donde A es el ancho de la corrugadora.

2.1 Modelo 1: Anillo-Estrella

Un *grafo anillo-estrella* es un grafo compuesto por un conjunto de ciclos (en este contexto llamados *anillos*) y tal que cada nodo o bien está dentro de un anillo o bien está unido por una arista a otro nodo que sí está dentro de un anillo (llamamos a esta arista una *conexión estrella*). La Figura 2.1 tiene un ejemplo de un *grafo 3-anillo-estrella* con tres anillos y cinco conexiones estrella.

Para esta formulacion se adaptó el modelo propuesto en [1] para el problema de m-anillo-estrella con capacidades. Es interesante observar que las soluciones factibles del PCTC tienen una estructura similar a los anillos-estrella estudiados en [1], a pesar de que se trata de problemas que surgen en contextos muy distintos. En efecto, una secuencia de combinaciones que fabrica en forma completa un conjunto de pedidos se corresponde con un camino en el grafo de combinaciones, eventualmente con algunos nodos unidos al camino por medio de

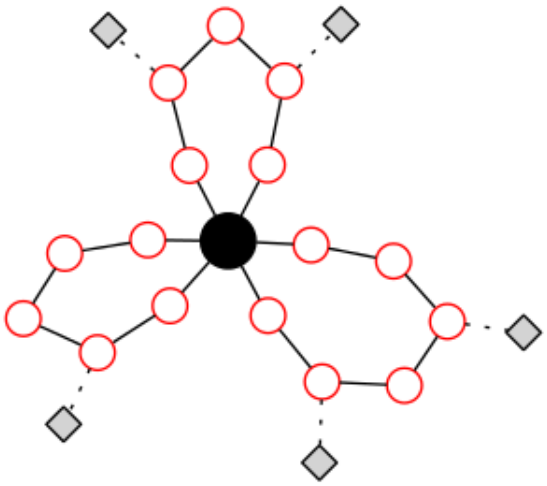

Figura 2.1: Grafo anillo-estrella con tres anillos y cinco conexiones estrella.

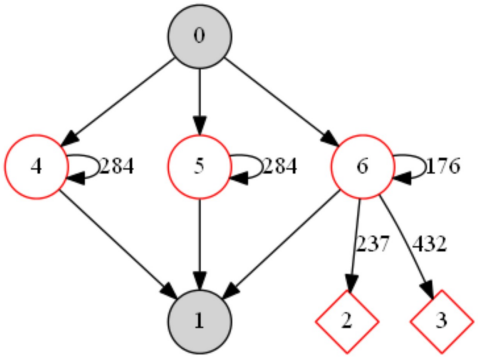

Figura 2.2: Grafo anillo-estrella asociado con la solución de la Figura 1.3

una conexión estrella. Esta estructura en el grafo determina que cada pedido se fabrica en un conjunto de combinaciones consecutivas, siempre que las combinaciones se fabriquen en el orden de recorrida del camino.

Por ejemplo, la Figura 2.2 muestra gráficamente la solución factible de la Figura 1.3 para la instancia de la Tabla 1.1. Al igual que el modelo presentado en [1], se generan dos nuevos nodos d_0 y d_1 que contienen el comienzo y el final de cada anillo. La solución óptima de este ejemplo induce un grafo con tres anillos con un pedido en cada uno y dos conexiones estrella conectados al pedido 6.

Para plantear el *modelo anillo-estrella*, introducimos las siguientes variables. Para $i \in P \cup \{d_0\}$ y $j \in P \cup \{d_1\}$ con $i \neq j$, la variable binaria x_{ij} toma el valor 1 si i y j están conectados de manera directa en un anillo, y 0 en caso contrario. En forma similar, para $i, j \in P$ con $i \neq j$, la variable binaria y_{ij} toma el valor 1 si i está conectado a j por una conexión estrella y 0 en caso contrario. Para evitar la aparición de anillos que no comiencen en d_0 o terminen en d_1, para $i \in P$ definimos la variable entera $z_i \in \{1, \ldots, |P|\}$ que contiene el número de orden del pedido i dentro del anillo al cual pertenece, salvo que esté unido al anillo por medio de una conexión estrella. Finalmente, para $i, j \in P$ y para $c \in \{1, \ldots, \gamma_{ij}\}$, consideramos la variable real $m_{ijc} \geq 0$ que representa los metros lineales por los cuales se

fabrica la c-ésima combinación que involucra los pedidos i y j. Con estas definiciones, podemos plantear el modelo de la siguiente forma.

1. La función objetivo solicita que se minimice el desperdicio total.

$$\text{mín} \sum_{i \in P} \sum_{j \in P} \sum_{c=1}^{\gamma_{ij}} m_{ijc} \, D_{ijc}$$

2. La primera restricción especifica que la cantidad de anillos que salen del nodo d_0 es la misma que la cantidad de anillos que llegan al nodo d_1.

$$\sum_{j \in P} x_{d_0,j} = \sum_{k \in P} x_{k,d_1}$$

3. Todo pedido está en algún anillo (es decir, recibe una conexión a través de una variable x desde otro pedido o bien desde d_0) o está conectado a un anillo por una conexión estrella (es decir, es el origen de una conexión estrella a través de una variable y con otro pedido).

$$\sum_{j \in P \cup \{d_0\}, i \neq j} x_{ji} + \sum_{j \in P, i \neq j} y_{ij} = 1 \qquad \forall i \in P$$

4. Todo pedido tiene un arco de entrada si y solo si tiene un arco de salida.

$$\sum_{j \in P \cup \{d_0\}, i \neq j} x_{ji} = \sum_{j \in P \cup \{d_1\}, i \neq j} x_{ij} \qquad \forall i \in P$$

5. Si j es un nodo estrella que está conectado con un nodo i a través de una conexión estrella (es decir, $y_{ji} = 1$), entonces i tiene que estar en un anillo.

$$y_{ji} \leq \sum_{k \in P \cup \{d_0\}} x_{ki} \qquad \forall i \in P, \forall j \in P, i \neq j$$

6. Las siguientes restricciones definen valores válidos para las variables z, y funcionan como restricciones de *rompimiento de subtours*.

$$z_j \geq (z_i + 1) - |P| \, (1 - x_{ij}) \qquad \forall i, j \in P, i \neq j$$

7. La cantidad de cajas producidas del pedido i tiene que estar dentro del rango permitido $[(1-p)q_i, (1+p)q_i]$.

$$q_i(1-p) \;\leq\; \sum_{j \in P} \sum_{c=1}^{\gamma_{ij}} n^i_{ijc} m_{ijc} / l_i \;\leq\; q_i(1+p) \qquad \forall \; i \in P$$

8. Si $m_{ijc} > 0$, entonces tiene que estar activada o bien una conexión anillo o bien una conexión estrella entre i y j.

$$MAX_{ijc} \, (x_{ij} + y_{ij}) \geq m_{ijc} \qquad \forall i, j \in P, i \neq j, c = 1, \ldots, \gamma_{ij}$$

9. Restricciones que especifican la naturaleza de las variables.

$$\begin{aligned}
x_{ij} &\in \{0,1\} \qquad \forall i \in P \cup \{d_0\}, \forall j \in P \cup \{d_1\} \\
y_{ij} &\in \{0,1\} \qquad \forall i, j \in P \\
z_i &\in \mathbb{Z}_+ \qquad \forall i \in P \\
m_{ijc} &\geq 0 \qquad \forall i, j \in P, \; c = 1, \ldots, \gamma_{ij}
\end{aligned}$$

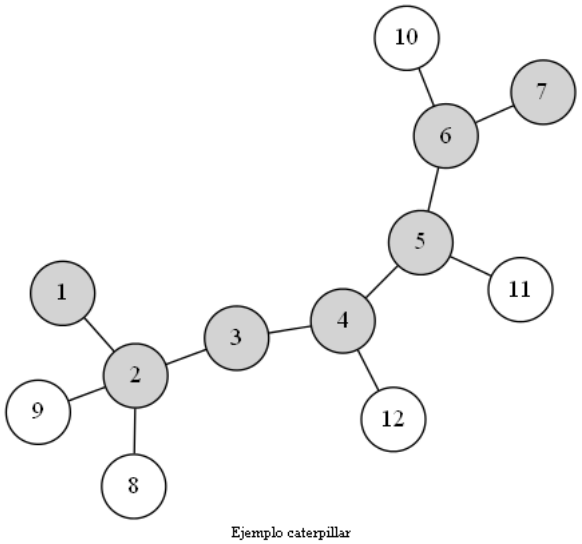

Figura 2.3: Un grafo caterpillar.

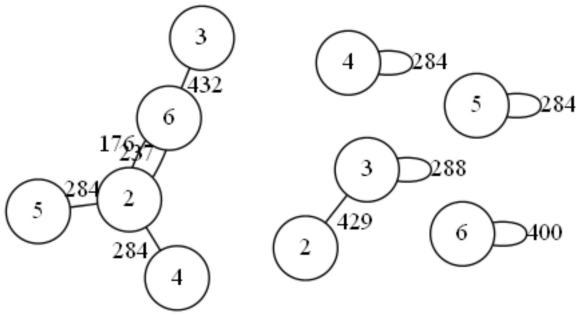

Figura 2.4: Dos soluciones factibles representadas como unión disjunta de grafos caterpillar para la instancia de la Figura 2.1

2.2 Modelo 2: Grafo caterpillar

Un grafo *caterpillar* (o grafo cienpiés) es un grafo tal que si se eliminan todos sus nodos de grado 1 (es decir, sus hojas) resulta en un camino o en un grafo vacío. Por ejemplo, la Figura 2.3 es un ejemplo de un grafo caterpillar.

En este modelo se formula el problema aprovechando que el grafo inducido por cualquier solución factible del problema es la unión disjunta de grafos caterpillar (exceptuando los pedidos autocombinados). Al igual que en el modelo anterior, cada combinación tiene una variable asociada que representa los metros lineales que se fabrican de la combinación, y si esta variable tiene un valor positivo entonces se "activa" la arista correspondiente del grafo de combinaciones. Las aristas activadas deben formar una unión disjunta de grafos caterpillar. Por ejemplo, los grafos de la Figura 2.4 corresponden a dos soluciones factibles para la instancia de ejemplo presentada en la Figura 2.1.

Decimos que un nodo $i \in P$ es un *nodo interno* si tiene grado mayor o igual a 2 en el grafo

inducido por la solución factible, y en caso contrario decimos que es una *hoja*. Para $i \in P$, definimos la variable binaria x_i que toma el valor 1 si el nodo i corresponde a un nodo interno y toma el valor 0 en caso contrario. En forma similar, para $i \in P$ definimos la variable binaria y_i que toma el valor 1 si el nodo i es una hoja y toma el valor 0 en caso contrario.

Para $i, j \in P$ con $i \neq j$, definimos la variable binaria e^x_{ij} de modo tal que $e^x_{ij} = 1$ si en el grafo inducido por la solución existe una arista entre i y j y además ambos nodos son nodos internos, y $e^x_{ij} = 0$ en caso contrario. En forma similar, para $i, j \in P$ con $i \neq j$, definimos la variable binaria e^y_{ij} de modo tal que $e^y_{ij} = 1$ si existe una arista entre i y j y además uno de los dos nodos es una hoja, y $e^y_{ij} = 0$ en caso contrario. Finalmente, para $i, j \in P$ y $c = 1, \ldots, \gamma_{ij}$, utilizamos la variable real $m_{ijc} \geq 0$ que representa los metros lineales fabricados de la c-ésima combinación entre los pedidos i y j.

1. Al igual que con todos los modelos para el problema, la función objetivo especifica que se debe minimizar el desperdicio total.

$$\text{mín} \sum_{i \in P} \sum_{j \in P} \sum_{c=1}^{\gamma_{ij}} m_{ijc}\, D_{ijc}$$

2. El primer grupo de restricciones especifica que todo nodo debe ser o bien una hoja o bien un nodo interno. Es importante notar que dentro de este modelo, los nodos que se autocombinan y están aislados en el grafo inducido por la solución pueden ser tanto nodos internos como hojas.

$$x_i + y_i = 1 \qquad \forall i \in P$$

3. Para $i, j \in P$ con $i \neq j$, la variable $e^y_{i,j}$ puede valer 1 sólo si i o j son hojas.

$$e^y_{ij} \leq y_i + y_j \qquad \forall i, j \in P, i \neq j$$

4. Para $i \in P$, si $y_i = 1$ entonces el nodo i es una hoja y por lo tanto debe tener grado 1.

$$y_i \leq \sum_{j \in P} e^y_{ij} \leq |P|\,(1 - y_i) + y_i \qquad \forall i \in P$$

5. Los nodos internos tienen como máximo grado 2 entre nodos internos. Dejamos libre la posibilidad de que un nodo i con $x_i = 1$ tenga grado menor que 2, dado que no afecta la factibilidad de las soluciones obtenidas e impone una menor restricción sobre la región factible.

$$\sum_{j \in P} e^x_{ij} \leq 2x_i \quad \forall i \in P, i \neq j.$$

6. El grafo inducido por la solución factible no puede tener ciclos entre los nodos internos. Para esto, se imponen las siguientes restricciones de *eliminación de subtours* entre los nodos internos. Llamamos $\mathcal{C}$ al conjunto de todos los ciclos de G, considerados como conjuntos de aristas.

$$\sum_{ij \in C} e^x_{ij} \leq |C| - 1 \qquad \forall C \in \mathcal{C}$$

7. Para $i, j \in P$ con $i \neq j$ y para $c = 1, \ldots, \gamma_{ij}$, si $m_{ijc} > 0$ entonces la arista ij debe formar parte del subgrafo de G inducido por la solución factible.

$$m_{ijc} \leq MAX_{ijc}(e^x_{ij} + e^y_{ij}) \quad \forall i, j \in P, c = 1, \ldots, \gamma_{ij}$$

8. La cantidad de cajas producidas de cada pedido tiene que estar dentro del rango permitido.

$$q_j(1-p) \leq \sum_{i \in P} \sum_{c=1}^{\gamma_{ij}} n^j_{ijc} m_{ijc}/l_j \leq q_j(1+p) \qquad \forall\ j \in P$$

9. Simetría para las variables $e^x_{i,j}$ y e^y_{ij}.

$$\begin{aligned} e^x_{ij} &= e^x_{ji} \quad \forall i, j \in P, i \neq j \\ e^y_{ij} &= e^y_{ji} \quad \forall i, j \in P, i \neq j \end{aligned}$$

10. Restricciones que especifican la naturaleza de las variables.

$$\begin{aligned} x_i &\in \{0,1\} \qquad \forall i \in P \\ y_i &\in \{0,1\} \qquad \forall i \in P \\ e^x_{ij} &\in \{0,1\} \qquad \forall i, j \in P,\ i \neq j \\ e^y_{ij} &\in \{0,1\} \qquad \forall i, j \in P,\ i \neq j \\ m_{ijc} &\geq 0 \qquad \forall i, j \in P,\ c = 1, \ldots, \gamma_{ij} \end{aligned}$$

2.3 Modelo 3: Caterpillar con cortes

El modelo planteado en esta sección se basa en la siguiente caracterización de los grafos caterpillar. Un *grafo garra bipartito* (bipartite claw) es un grafo de 7 nodos isomorfo al grafo de la Figura 2.5.

Teorema 1. *(Lekkerkerker y Boland, 196x [9]) Un grafo conexo G es un grafo caterpillar si y sólo si*

1. *G no tiene ciclos.*
2. *G no contiene ningún* grafo garra bipartito *como subgrafo inducido.*

En este modelo sólo utilizamos variables binarias x_{ij} para cada $i, j \in P$ con $i \neq j$, de modo tal que $x_{ij} = 1$ si la solución utiliza alguna combinación que involucra los pedidos i y j, y $x_{ij} = 0$ en caso contrario. Además, para cada $i, j \in P$ y para $c = 1, \ldots, \gamma_{ij}$, utilizamos la variable real $m_{ijc} \geq 0$ que representa los metros lineales que se fabrican de la c-ésima combinación entre los pedidos i y j.

1. La función objetivo especifica que se debe minimizar el desperdicio total.

$$\text{mín} \sum_{i \in P} \sum_{j \in P} \sum_{c=1}^{\gamma_{ij}} m_{ijc}\, D_{ijc}$$

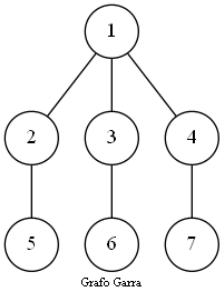

Figura 2.5: Un grafo garra bipartito.

2. El primer grupo de restricciones especifica que el subgrafo de G inducido por las variables x activas no debe tener ciclos.

$$\sum_{ij \in C} x_{ij} \leq |C| - 1 \qquad \forall\ C \in \mathcal{C}$$

3. Sea $\mathcal{B}$ el conjunto de todas las garras bipartitas de G (no necesariamente inducidas), consideradas como conjuntos de aristas. El siguiente grupo de restricciones especifica que el subgrafo de G inducido por las variables x activas no debe tener garras bipartitas.

$$\sum_{ij \in B} x_{ij} \leq 5 \qquad \forall\ C \in \mathcal{B}$$

4. Las siguientes restricciones especifican el vínculo entre las variables x y m, de modo tal que $x_{ij} = 1$ si $m_{ijc} > 0$ para algún $c = 1, \ldots, \gamma_{ij}$.

$$m_{ijc} \leq MAX_{ijc}\, x_{ij} \qquad \forall\ i, j \in P, i \neq j,\ c = 1, \ldots, \gamma_{ij}$$

5. Se fabrica la cantidad solicitada de cada pedido.

$$q_j(1-p) \leq \sum_{i \in P} \sum_{c=1}^{\gamma_{ij}} n_{ijc}^{j} m_{ijc}/l_j \leq q_j(1+p) \qquad \forall\ j \in P$$

6. Las siguientes restricciones expresan que las variables x_{ij} y x_{ji} se consideran indistintamente, para $i, j \in P$.

$$x_{ij} = x_{ji} \quad \forall i, j \in P$$

7. Restricciones que especifican la naturaleza de las variables.

$$\begin{aligned} x_{ij} &\in \{0,1\} \qquad \forall i, j \in P,\ i \neq j \\ m_{ijc} &\geq 0 \qquad \forall i, j \in P,\ c = 1, \ldots, \gamma_{ij} \end{aligned}$$

2.4 Modelo 4: Cuadrático

Sea C el conjunto de todas las combinaciones posibles. Dado un pedido $j \in P$, llamamos $C(j) \subseteq C$ al conjunto de las combinaciones que incluyen al pedido j. Llamamos t a la cantidad máxima de combinaciones que están presentes en la solución, y definimos $T = \{1, \ldots, t\}$.

En este modelo representamos explícitamente el orden de las combinaciones en el programa. Para esto, para cada $i \in C$ y cada $k \in T$ se introduce una variable binaria x_{ik} de modo tal que $x_{ik} = 1$ si la combinación i es la k-ésima combinación del programa de cortes, y $x_{ik} = 0$ en caso contrario. Además, para cada $i \in C$ se tiene una variable real $m_i \geq 0$ que representa los metros lineales que se fabrican de la combinación i.

1. La función objetivo especifica que se debe minimizar el desperdicio total. Para $i \in C$, definimos D_i como el desperdicio por metro lineal de la combinación i.

$$\text{mín} \sum_{i \in C} m_i \, d_i$$

2. El primer grupo de restricciones especifica que sólo una combinación es la k-ésima dentro del programa de cortes.

$$\sum_{i \in C} x_{ik} \leq 1 \qquad \forall \; k \in T$$

3. Los metros lineales de una combinación son cero si la combinación no se usa en el programa de cortes. Para $i \in C$, llamamos MAX_i a una cota superior de los metros lineales que se pueden fabricar de la combinación i, calculada de la misma forma que en los modelos anteriores.

$$m_i \leq MAX_i \sum_{k \in T} x_{ik} \qquad \forall \; i \in C$$

4. Se fabrica la cantidad solicitada de cada pedido, donde prod_{ij} representa la cantidad de planchas fabricadas del pedido j por cada metro lineal de la combinación i, para $i \in C$ y $j \in P$:

$$q_j(1-p) \leq \sum_{i \in C(j)} \text{prod}_{ij} \, m_i \leq q_j(1+p) \qquad \forall \; j \in P$$

5. Los tramos en los que participa un pedido son consecutivos. Si hay producción del pedido j en el tramo l, entonces la falta de producción en el tramo $k-1$ (con $k-1 > l$) implica que no hay producción en el tramo k para este pedido:

$$\sum_{i \in C(j)} x_{ik} \leq \sum_{i \in C(j)} x_{i,k-1} + M\Big(1 - \sum_{i \in C(j)} x_{il}\Big) \qquad \forall \; j \in P, \; \forall \; k, l \in T, \; l < k-1$$

6. Restricciones que especifican la naturaleza de las variables.

$$\begin{aligned} x_{ik} &\in \{0,1\} \qquad \forall i \in C, \; \forall k \in T \\ m_i &\geq 0 \qquad \forall i \in C \end{aligned}$$

2.5 Comparaciones

En esta sección se comparan los cuatro modelos para 34 instancias de diversa dificultad. Las instancias utilizadas fueron extraidas de programas de producción reales y ordenadas en forma creciente de acuerdo a la cantidad de pedidos que tiene cada una, siendo la más pequeña de cinco pedidos diferentes con un promedio de 1100 planchas y la instancia más grande de 55 pedidos con un promedio de 4038 planchas cada uno.

Si bien ningún pedido supera el ancho de la corrugadora, existen pedidos de ancho igual al ancho total, lo que hace imposible que se puedan combinar con otros pedidos. Alrededor de 600 pedidos tienen un ancho menor a 500 mm, 20 pedidos tienen anchos entre 500 mm y 1000 mm, y 290 pedidos tienen anchos entre 1000 mm y 1400 mm. Las instancias incluyen un total de 267988 metros cuadrados, con un promedio de 14482 metros cuadrados por instancia donde la instancia más pequeña requiere procesar 818 metros cuadrados y la más grande involucra 38723 metros cuadrados.

Para los experimentos se utilizó un ancho máximo de corrugadora de 1400 mm y un desperdicio en la cantidad total de planchas tolerable p del 5 %. Los experimentos se realizaron en una computadora con procesador *AMD FX(tm)-6100 Six-Core*, 8GB de memoria RAM y con un sistema operativo de 64 bits. Se utilizó el paquete *cplex* (*CPLEX Optimization Studio Academic Research Edition 12.3*).

Los modelos *caterpillar* y *caterpillar con cortes* involucran restricciones de rompimiento de ciclos que están compuestas por un número exponencial de desigualdades. Además, el modelo *caterpillar con cortes* incluye restricciones de rompimiento de garras bipartitas, compuestas por $O(n^7)$ desigualdades. En nuestra implementación, ambas restricciones se agregan dinámicamente durante la ejecución del algoritmo *branch and bound*. Se generan estas restricciones después de que se encuentra una solución entera, si el grafo resultante no cumple con ser *caterpillar*. El procedimiento de separación de estos algoritmos es similar a los algoritmos de separación en las familias de desigualdades *ciclo* y *garra* que se verán en el próximo capitulo: se genera un grafo utilizando las variables asociadas a los ejes del grafo de combinaciones con valores enteros iguales a 1, y se recorre este grafo en busca de *ciclos* o *garras*. En caso de encontrar estas estructuras, se agregan las restricciones asociadas como *lazy constraints* dentro de *cplex*.

2.5.1 Análisis de los tiempos de ejecución

En el Cuadro 2.1 se muestran los resultados de las corridas de los cuatro modelos para diferentes refiles (5 %, 10 % y 100 %). Las diferencias entre las configuraciones de los refiles influyen en la cantidad de combinaciones que se deben tener en cuenta: si el refile permitido es pequeño, entonces las combinaciones entre pedidos están más limitadas. Teniendo en cuenta esto, se eligieron valores para el refile que generan casos donde tenemos pocas, intermedias y muchas combinaciones. Es importante notar que si el refile es demasiado pequeño muchas instancias resultan *no factibles* (se muestran subrayadas en el cuadro), mientras que si el refile máximo permitido es alto todas las instancias resultan *factibles*. Las corridas marcadas con "*Unknown*" corresponden a casos en los que no se llegó a la solución óptima luego de una hora de ejecución. Para el caso del modelo *cuadratico* notar que hay muchas instancias marcadas con "-", esto representa que para dichas instancias no se ha podido generar en memoria el modelo de programación lineal entera, debido a su gran tamaño.

Queda en evidencia que el modelo *caterpillar con cortes* demora mucho tiempo. Sin em-

bargo, el modelo *caterpillar con cortes* no es el modelo con peores resultados teniendo en cuenta que para el modelo *cuadrático*, a partir de la instancia 14, no se han podido generar los modelos iniciales correpondientes en tiempos razonables superando en todos los casos la memoria física de la máquina. Este resultado era esperable debido a la cantidad de combinaciones que se deben procesar durante la generación del modelo inicial.

Los modelos *anillo-estrella* y *caterpillar* fueron bastantes parejos entre sí y presentaron mejores resultados. Sin embargo, el modelo *caterpillar* fue capaz de resolver más instancias que el modelo *anillo-estrella*. Por otro lado, cabe mencionar que el promedio de respuesta para las pruebas realizadas con un refile del 100 % es mayor para el modelo *caterpillar* (teniendo en cuenta que existe un outlier en la instancia 29), sin embargo, el modelo *anillo-estrella* no pudo llegar a resolver todas las instancias en el tiempo límite propuesto. Por lo tanto, se considera el modelo *caterpillar* para la siguiente etapa del trabajo.

	Refile 5 %				Refile 10 %				Refile 100 %			
Instancia	Anillo-Estrella	Caterpillar	Caterpillar con cortes	Cuadrático	Anillo-Estrella	Caterpillar	Caterpillar con cortes	Cuadrático	Anillo-Estrella	Caterpillar	Caterpillar con cortes	Cuadrático
Instancia 1	0.016	0.016	10.016	0.015	0	0.016	10.016	0.031	0.031	0.016	0.016	0.047
Instancia 2	0.015	0.015	10	0.015	0.015	0.016	10	0.015	0.015	0.016	0.015	0.032
Instancia 3	0	0	0	0	0.015	0.016	20.015	0.047	0.016	0.016	0.016	0.047
Instancia 4	0.016	0	0.015	0	0.015	0.015	0	0.016	0.078	0.031	0.016	0.187
Instancia 5	0.015	0.016	10	0.062	0.031	0.016	40.047	0.109	0.063	0.047	0.015	0.359
Instancia 6	0	0.016	0.015	0.015	0	0.016	0	0.047	0.047	0.031	0.047	0.203
Instancia 7	0	0	0	0	0	0	0	0.031	0.093	0.062	0.032	5.601
Instancia 8	0.046	0.062	50.063	0.499	0.047	0.031	30.094	1.497	0.109	0.156	0.655	3.728
Instancia 9	0	0	0	0	0	0	0	0.015	0.062	0.063	0.047	2.168
Instancia 10	0	0	0.015	0.016	0.094	0.078	50.11	5.32	0.218	0.187	0.032	18.689
Instancia 11	0.234	0.078	30.047	14.991	0.14	0.124	110.187	44.569	0.406	0.562	0.078	1464.428
Instancia 12	0.062	0.047	0.047	0	0.452	1.716	1832.807	1839.985	1.622	5.132	478.065	-
Instancia 13	0.063	0.047	0.031	1.232	0.078	0.047	0.031	0.015	0.904	5.429	344.949	-
Instancia 14	0	0.016	0.031	0.047	0.032	0.031	0.015	2.309	0.218	2.481	11.654	129.247
Instancia 15	0	0.015	0	0.249	1.42	3.432	*Unknown*	3371.572	5.538	17.269	3600.035	-
Instancia 16	0.39	1.248	233.573	*Unknown*	0.406	0.546	173.385	45.88	0.437	1.216	1.856	-
Instancia 17	0	0.016	0.015	0.109	0.499	0.748	181.45	228.619	0.64	16.911	19.36	-
Instancia 18	0.125	0.047	0.015	-	5.554	1.373	*Unknown*	3619.426	1.482	13.275	355.105	-
Instancia 19	27.067	20.514	*Unknown*	-	8.331	11.388	34425.227	-	12.917	49.39	3600.737	-
Instancia 20	1.935	7.862	*Unknown*	-	2.215	2.434	*Unknown*	-	7.301	9.844	3600.176	-
Instancia 21	2.574	8.923	2773.086	-	10.857	14.196	*Unknown*	-	7.597	83.71	223.94	-
Instancia 22	1.061	0.343	564.491	-	0.921	0.515	*Unknown*	-	2.215	3.198	135.83	-
Instancia 23	0	0.015	0.031	-	0	0	0	1.279	0.702	5.429	647.42	-
Instancia 24	0.343	0.374	51.201	-	0.578	1.435	201.591	-	0.687	14.711	58.079	-
Instancia 25	5.414	12.371	*Unknown*	-	10.109	10.343	*Unknown*	-	19.687	172.366	*Unknown*	-
Instancia 26	*Unknown*	0.936	937.206	-	*Unknown*	1.529	1561.121	-	*Unknown*	13.572	2579.992	-
Instancia 27	*Unknown*	5.304	*Unknown*	-	*Unknown*	94.24	*Unknown*	-	*Unknown*	148.31	*Unknown*	-
Instancia 28	32.042	30.998	*Unknown*	-	50.42	50.638	*Unknown*	-	472.261	516.067	*Unknown*	-
Instancia 29	698.635	67.486	*Unknown*	-	3268.376	152.585	*Unknown*	-	*Unknown*	3312.666*	*Unknown*	-
Instancia 30	17.191	10.545	*Unknown*	-	23.946	28.283	*Unknown*	-	203.784	194.049	*Unknown*	-
Instancia 31	0.328	0.281	0.047	-	33.93	36.286	*Unknown*	-	274.577	236.186	*Unknown*	-
Instancia 32	9.235	3.026	*Unknown*	-	16.256	8.237	*Unknown*	-	71.058	46.192	*Unknown*	-
Instancia 33	0.312	0.094	0.063	-	22.605	20.935	*Unknown*	-	*Unknown*	497.69	*Unknown*	-
Instancia 34	10.42	16.911	*Unknown*	-	19.344	35.739	*Unknown*	-	280.614	287.307	3600.347	-
Promedio	**47.44**	**9.84**	**186.79912**	**3.11**	**139.06**	**17.66**	**1932.30**	**654.07**	**44.04**	**68.83**	**713.27**	**125.00**

Cuadro 2.1: Tiempo de ejecución de los modelos para diferentes refiles

* Teniendo en cuenta que esta misma instancia no pudo resolverse para el resto de los modelos, este valor es ignorado en el promedio final.

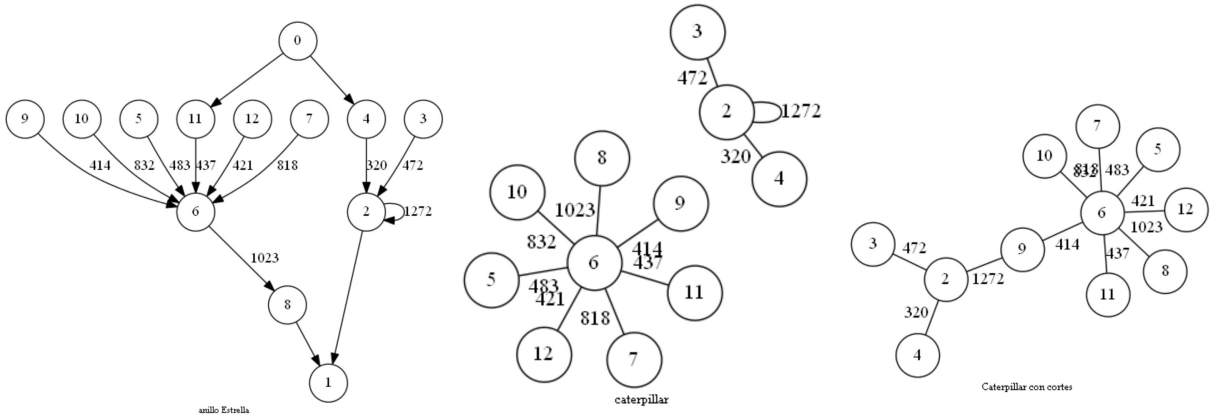

Figura 2.6: Grafos para la instancia 9

2.5.2 Características de las soluciones obtenidas

Durante los experimentos computacionales se observó que cada modelo tiende a generar soluciones óptimas con características distintas (que corresponden a óptimos alternativos). En esta sección analizamos empíricamente la estructura del *grafo solución*, cuyos nodos corresponden a los pedidos y cuyas aristas corresponden a las combinaciones seleccionadas. Es interesante mencionar algunos detalles encontrados durante el análisis que tienen que ver con la solución gráfica para estos modelos. Se estudian los modelos *anillo estrella*, *caterpillar* y *caterpillar con cortes*.

Los grafos del modelo *anillo estrella* son grafos conexos, en los que existe un camino desde el nodo 0 al nodo 1 pasando por el nodo i (donde i representa un pedido). Para este análisis tuvimos en cuenta el grafo resultado de quitar dichos nodos sabiendo que a efectos prácticos estos nodos son auxiliares dentro del modelo y no corresponden a la solución del problema.

Comparando los grafos solución para cada instancia se puede observar que no necesariamente son isomorfos, de hecho, dentro de la muestra que estudiamos se puede ver que no existen instancias donde el grafo solución sea el mismo en los tres casos. No obstante, los grafos del modelo *caterpillar* y *anillo estrella* tienden a ser más parecidos que el grafo para el modelo *caterpillar con cortes*. Por ejemplo, en la Figura 2.6 se muestran los grafos para los modelos en la instancia 9, se puede observar que la solución es la misma para los modelos *caterpillar* y *anillo estrella* mientras que para el modelo *caterpillar con cortes* es diferente.

Es interesante analizar la cantidad de componentes conexas presentes en las soluciones. Para el modelo *caterpillar con cortes* la cantidad de componentes conexas suele ser menor, mientras que para el modelo de *anillo estrella* la cantidad suele ser mayor. Los modelos *anillo estrella* tienden a crear más nodos aislados que el resto de los modelos, mientras que el modelo *caterpillar con cortes* tiende a crear grafos con caminos más largos. Como ejemplo de esto, las Figuras 2.7, 2.8 y 2.9 contienen los grafos solución para los modelos *caterpillar*, *caterpillar con cortes* y *anillo-estrella* respectivamente para una instancia relativamente grande (instancia 19) de 41 pedidos con 132570 cajas.

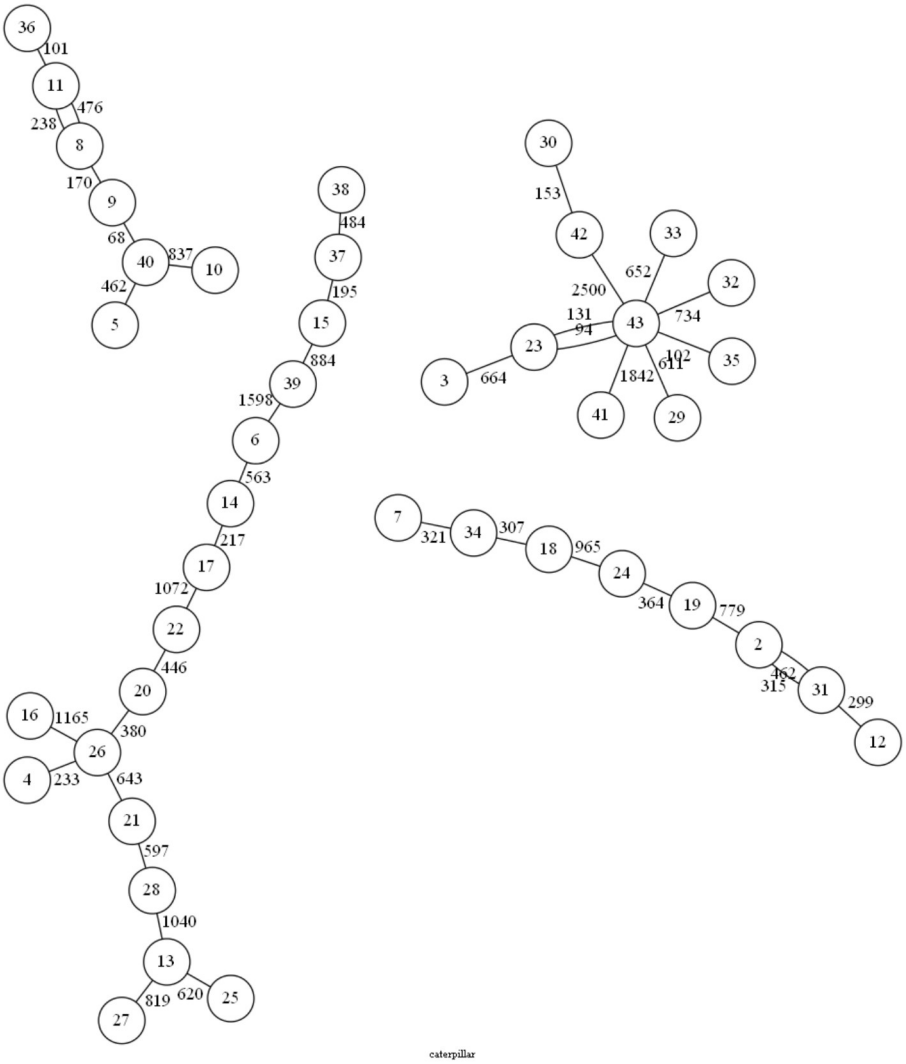

Figura 2.7: Solución del modelo *caterpillar* para la instancia 19

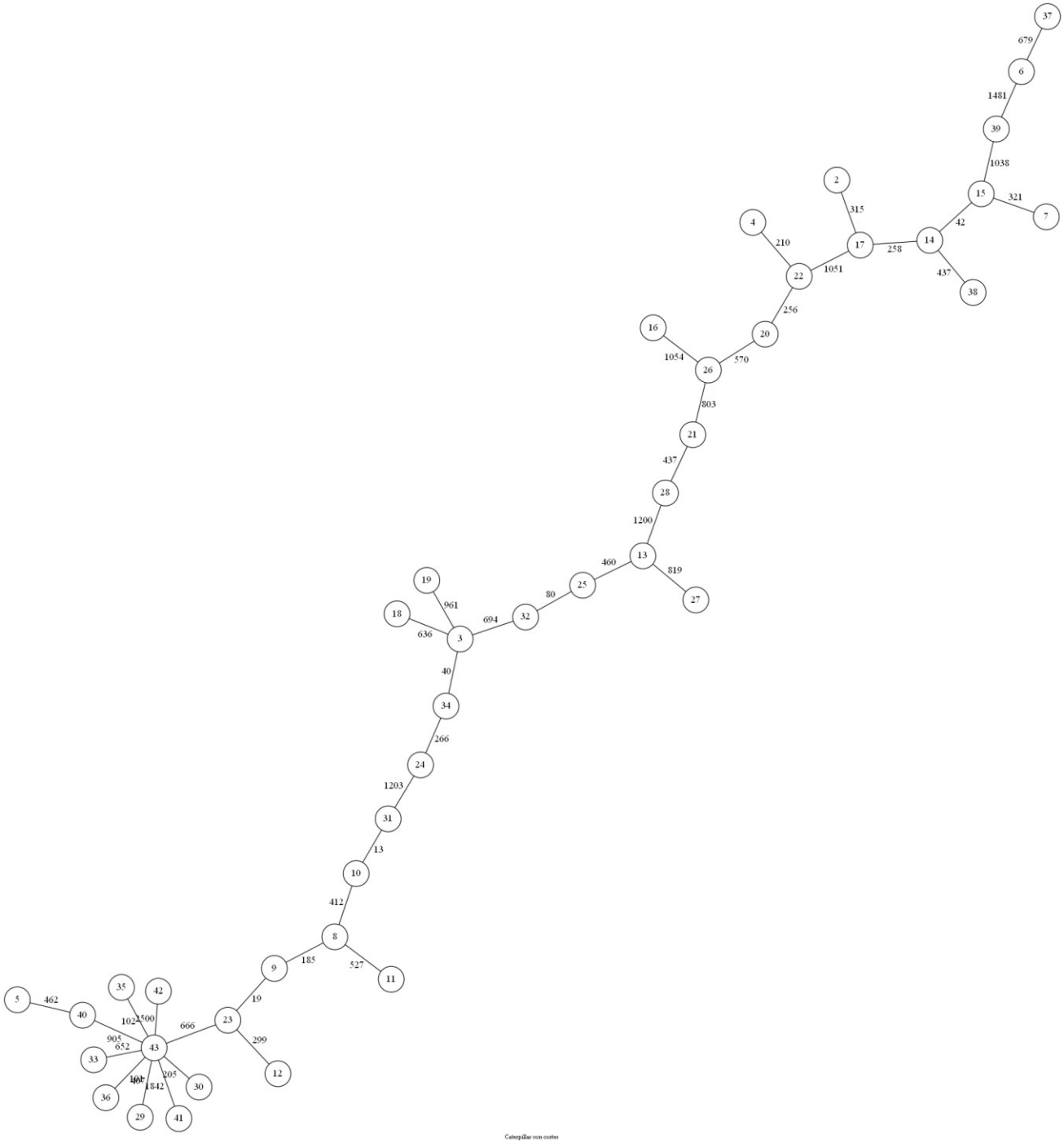

Figura 2.8: Solución del modelo *caterpillar con cortes* para la instancia 19

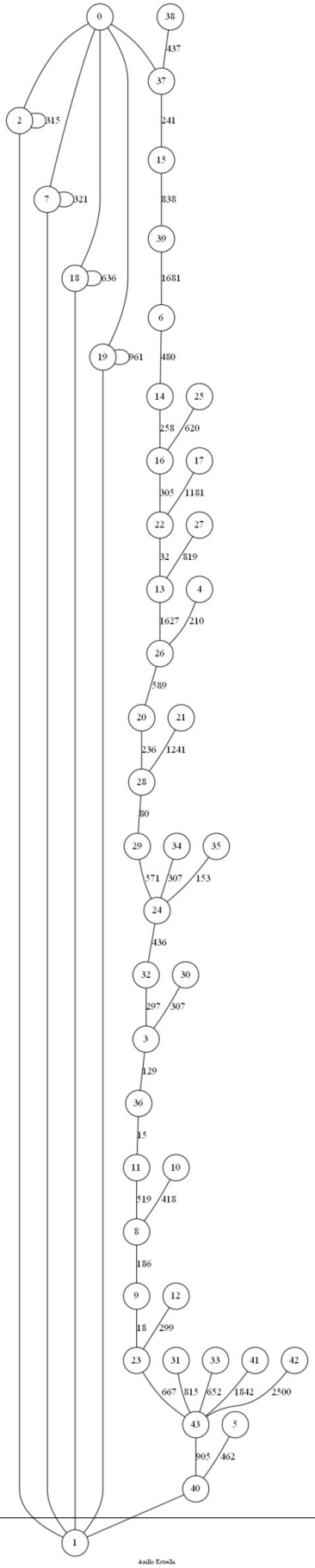

Figura 2.9: Solución del modelo *anillo-estrella* para la instancia 19

Capítulo 3

Desigualdades válidas

Debido a que el problema es difícil buscamos conocimiento específico que pueda ayudar a *CPLEX*[7] a mejorar sus tiempos de ejecución. Ese conocimiento puede estar dado por familias de desigualdades válidas que se puedan usar como cortes durante la ejecución del algoritmo *branch and bound* o bien por heurísticas que puedan ayudar a mejorar las cotas primales durante el algoritmo.

En este capítulo se describen familias de desigualdades válidas para el modelo *caterpillar*, que dan origen a los planos de cortes usados en un algoritmo de tipo *branch and cut* para el problema. Para cada familia, se detalla un procedimiento de separación y se exponen los resultados computacionales.

Recordemos que el modelo *caterpillar* se basa en la formulación del problema modelado con grafos *caterpillar* cuyos ejes son representados por las variables e^x_{ij} y e^y_{ij}, las variables y_i representan nodos que son hojas, las variables x_i representan los nodos internos y las variables m_{ijc} representan los metros lineales para la c-ésima combinación de los pedidos i y j.

3.1 Familias de desigualdades válidas

3.1.1 Desigualdades Garra

Esta familia de desigualdades se basa en el hecho de que un grafo caterpillar no puede tener garras bipartitas. Al introducir estas desigualdades como cortes en un algoritmo *branch and cut*, buscamos que las soluciones intermedias vayan adoptando la forma de un caterpillar lo más temprano posible. Decimos que un nodo es un *nodo garra* si cumple que tiene tres vecinos y cada vecino tiene a su vez al menos otro vecino (ver la Figura 3.1).

La familia de desigualdades *garra* especifica que no se deben formar estas estructuras en el grafo solución, pidiendo que en todo conjunto de seis nodos que forme una garra bipartita no haya más de cinco ejes de la garra. Formalmente, esta familia está compuesta por las siguientes desigualdades válidas:

$$e^x_{ij} + e^y_{ij} + e^x_{il} + e^y_{il} + e^x_{ik} + e^y_{ik} + e^x_{jz} + e^y_{jz} + e^x_{lg} + e^y_{lg} + e^x_{km} + e^y_{km} \leq 5$$
$$\forall i, j, l, k, z, g, m \in P \quad y \quad i \neq j \neq z, i \neq k \neq m, i \neq l \neq g. \quad (3.1)$$

Otra manera de evitar nodos garra es pedir que exista al menos un eje e^y como vecino del nodo garra, teniendo en cuenta que si existe un eje de este tipo entonces el nodo es un nodo

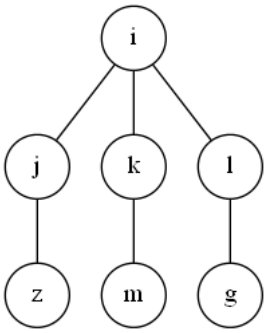

Figura 3.1: Nodo Garra

hoja. De esta forma se asegura que dicho nodo no tenga más de un vecino. Siguiendo estas ideas, podemos plantear la siguiente familia de desigualdades válidas:

$$y_j + y_k + y_l \geq e^x_{ij} + e^y_{ij} + e^x_{ik} + e^y_{ik} + e^x_{il} + e^y_{il} - 2$$
$$\forall j, k, l, \in P \quad y \quad i \neq j \neq k \neq l. \quad (3.2)$$

La cantidad de desigualdades que se pueden derivar tanto de esta familia de desigualdades como de la anterior presentada depende de la cantidad de garras y la cantidad de vecinos que tenga cada nodo garra. Por ejemplo, si u es un nodo garra y $v_1, v_2, ..., v_n$ son nodos adyacentes a u en la solución y cada v_i tiene un nodo adyacente w_i, entonces la cantidad de desigualdades garras para esta estructura es $O(n^3)$.

3.1.2 Desigualdades Ciclo

Al igual que la familia de desigualdades *garra* esta familia se basa en las propiedades que debe cumplir un grafo caterpillar. Teniendo en cuenta que en grafos de este tipo no pueden existir ciclos se plantea la siguiente familia de desigualdades (donde $\mathcal{C}$ representa el conjunto de todos los ciclos del grafo de pedidos, considerados como conjuntos de aristas):

$$\sum_{ij \in C} e^x_{ij} \leq |C| - 1 \qquad \forall C \in \mathcal{C} \quad (3.3)$$

Cabe señalar que estas desigualdades son parte del modelo original y se agregan como restricciones dentro de la ejecución del algoritmo cuando se encuentra una solución con todas las variables binarias con valores enteros pero que no es factible porque contiene un ciclo. Sin embargo, también se puede realizar la separación de estas desigualdades sobre las soluciones fraccionarias que se obtienen al resolver los subproblemas del árbol de enumeración. Dado que resulta interesante evaluar el impacto de la generación dinámica de estos planos de corte también sobre las soluciones fraccionarias, consideramos estas restricciones como una nueva familia de desigualdades válidas. Cuando se menciona en los experimentos computacionales que se separan estas desigualdades, se refiere al hecho de que se aplica el algoritmo de separación mencionado en la siguiente sección sobre las soluciones fraccionarias, además de la separación de estas restricciones (y su agregado como *lazy constraints*) sobre las soluciones enteras pero no necesariamente factibles halladas a lo largo del algoritmo *branch and cut*.

3.1.3 Desigualdades Vecinos

Esta familia de desigualdades se basa en que en toda componente conexa del grafo asociado con la solución tienen que existir al menos dos hojas, por lo tanto, podemos pedir que existan dos ejes e^y_{ij}. Esto nace del hecho de que un grafo caterpillar es como mínimo un camino (si asumimos que un grafo de un nodo es un camino), por lo tanto tiene al menos dos hojas que son representadas por las variables y, como existen nodos hoja y por la definición de las variables tambien tienen que existir "ejes hoja" dados por las variables e^y. La siguiente expresión corresponde a estas observaciones, para el caso en el que C es un conjunto de *nodos internos*. En este caso, la desigualdad pide que la cantidad de ejes con origen en C y destino fuera de C sea como minimo 2.

$$\sum_{i \in C} \sum_{j \notin C} e^x_{ij} + e^y_{ij} \geq 2(\sum_{i \in C} x_i - |C| + 1) \qquad \forall C \subseteq P. \tag{3.4}$$

En este caso C representa una componente conexa dentro del grafo de las soluciones intermedias. Esta familia está compuesta por $O(2^n)$ desigualdades, donde n es la cantidad de pedidos.

3.1.4 Desigualdades para los nodos internos y externos

Esta familia de desigualdades impone que si existe un nodo interno x_i entonces tiene que existir un eje interno e^x_{ij} para algún otro pedido j. Esta restricción nace de observar cómo se comportan los valores de las variables en las soluciones fraccionarias que aparecen a lo largo del *branch and bound*. Se pudo observar empíricamente que las variables correspondientes a los ejes (e^x y e^y) se mantenían más tiempo en valores fraccionarios, mientras que las variables x e y llegaban a valores enteros más rápidamente. La intención de esta familia es que las variables que se corresponden a los ejes se vuelvan enteras más rápidamente en función de las variables x e y:

$$\sum_{j \in P} e^x_{ij} \geq x_i \qquad \forall i \in P. \tag{3.5}$$

Podemos repetir esta idea para los nodos hoja. En este caso, si existe algún nodo hoja y entonces tiene que existir algun "eje hoja", dado por las variables e^y:

$$\sum_{j \in P} e^y_{ij} \geq y_i \qquad \forall i \in P. \tag{3.6}$$

Ambas familias están compuestas por n desigualdades, siendo n la cantidad de pedidos.

3.2 Algoritmos de separación

Dada una solución fraccionaria, el *problema de separación* asociado con una familia de desigualdades válidas consiste en determinar si la solución viola alguna desigualdad de la familia. Un *algoritmo de separación* es un procedimiento exacto o heurístico para resolver el problema de separación asociado con una familia. Este tipo de algoritmos tiene un rol muy importante durante un algoritmo *branch and cut* porque podría suceder que una familia de desigualdades sea muy eficaz ajustando la relajación lineal pero si el algoritmo utilizado para separar esta familia no es efectivo, entonces se agregarán pocos cortes de esta familia y el impacto sobre

el procedimiento total será muy bajo o incluso negativo, si el tiempo de ejecución del algoritmo de separación es alto. Por tal motivo, estos procedimientos tienen que ser eficientes. En muchas ocasiones se utilizan heurísticas, buscando un balance entre el costo computacional y la cantidad de cortes hallados.

En esta sección proponemos algoritmos de separación para las familias de desigualdades introducidas en la sección anterior. Todos los algoritmos mencionados en esta sección se encuentran en pseudocódigo en el Anexo A.3.4.

La mayoría de los algoritmos de esta sección calcula primero un grafo intermedio a partir de la solución fraccionaria, utilizando una cota inferior k para las variables e^y y e^x. Los nodos de este grafo están dados por los pedidos, y se agrega un "eje y" (respectivamente, un "eje x") entre los nodos i y j si $e^y_{ij} \geq k$ (respectivamente, si $e^x_{ij} \geq k$). El valor de k determina indirectamente la densidad de este grafo. La construcción de este grafo se puede realizar en tiempo $O(n^2)$, donde n es la cantidad de pedidos.

Una vez armado este grafo, se busca si existen desigualdades válidas de la familia violadas por el punto fraccionario en cuestión.

- El algoritmo de separación de las desigualdades *garra* (3.1 y 3.2) para cada componente conexa busca los nodos garra y luego, utilizando un algoritmo recursivo, quita en cada paso un eje vecino del nodo garra para evitar que en la próxima iteración vuelva a caer en la misma garra. Este algoritmo es cuadrático en la cantidad de ejes de la componente conexa. Una vez detectados los ejes involucrados en una garra, se arma una nueva restricción pidiendo que la suma de los ejes sea menor a seis o que los nodos adyacentes al nodo garra sean nodos hojas (y), si dichas restricciones son violadas se agregan como cortes al problema original. Este algorimo finaliza una vez inspeccionados todos los nodos en busca de nodos *garra*, es decir, en esta instancia, se agregan tantas restricciones nuevas como garras en el grafo existan.

- Para la familia de desigualdades *ciclo* (3.3) se calcula para cada componente conexa la cantidad de nodos y utilizando las variables asociadas a los ejes se construye la desigualdad asociada y se chequea si está violada. La complejidad de este algoritmo es lineal en la cantidad de ejes del grafo.

- El procedimiento de separación para la familia de desigualdades *vecinos* (3.4) también se basa en este grafo intermedio. En este caso, para cada componente conexa se analizan las variables e^x, e^y y x y se determina si la desigualdad asociada con la componente conexa está violada. El algoritmo es lineal en la cantidad de nodos.

- Por último, para separar las familias de desigualdades para los *nodos internos* (3.5 y *nodos externos* 3.6) no se utiliza el grafo intermedio descripto arriba, sino que se inspeccionan todas las desigualdades de la familia iterando sobre todos los pedidos y reteniendo las desigualdades violadas.

3.3 Experiencia computacional

En esta sección se presentan los resultados de la adición de cortes generados a lo largo de la ejecución del algoritmo, convirtiendo de esta forma el esquema *branch and bound* en un algoritmo *branch and cut*. Se comparan los tiempos de respuesta de este nuevo algoritmo con el *branch and bound* original.

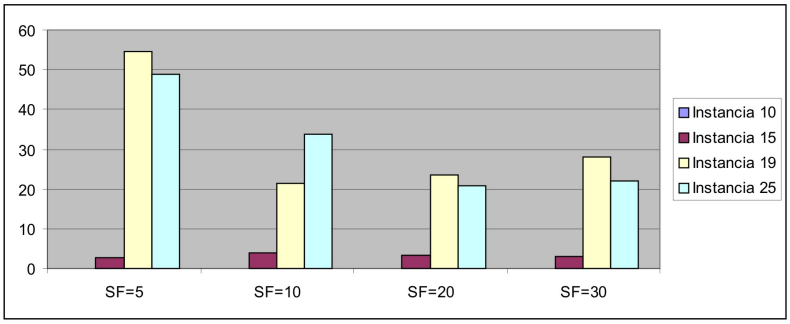

Figura 3.2: Tiempos de ejecución para algunas instancias en función del skip factor, para el algoritmo *branch and cut* con las desigualdades *garra*.

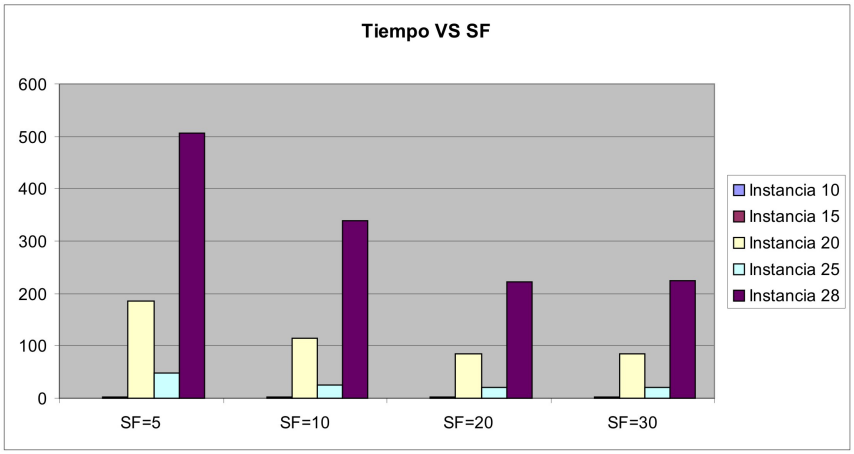

Figura 3.3: Tiempos de ejecución para algunas instancias en función del skip factor, para el algoritmo *branch and cut* con las desigualdades *ciclo*.

Como se mencionó anteriormente, un factor importante a considerar durante la experimentación es la complejidad del algoritmo de separación de cada familia. Teniendo en cuenta esto se creó un parametro *skip factor* (SF) que determina la frecuencia de aplicado de cada corte. Por ejemplo, el valor $SF = 5$ especifica que se ejecuta el algoritmo de separación cada cinco subproblemas del árbol de enumeración en el algoritmo *branch and cut*. De esta forma, se busca un balance entre el tiempo de separación y su efectividad.

Para determinar este valor se hicieron pruebas para instancias de diferentes tamaños. Las Figuras 3.2, 3.3 y 3.4 muestran estos resultados para la familia de desigualdades *garra* (3.1 y 3.2), *ciclo* (3.3) y *vecinos* (3.4) respectivamente. De estos resultados se desprende que cuanto mayor es la frecuencia de agregado de corte la performance se degrada más. Si los cortes se aplican muy seguido el tiempo desperdiciado en armar el grafo y buscar los planos de cortes degrada bastante los tiempos y, como los cortes no son efectivos, este tiempo perdido no se recupera. Para estas familias de desigualdades se utilizó durante los experimentos subsiguientes el valor $SF = 30$.

Para la familia de desigualdades *nodos* (3.5 y 3.6) no hay grandes cambios para los diferentes valores de SF, como muestra la Figura 3.5. En este caso, también se utilizó el valor $SF = 30$ en los experimentos subsiguientes.

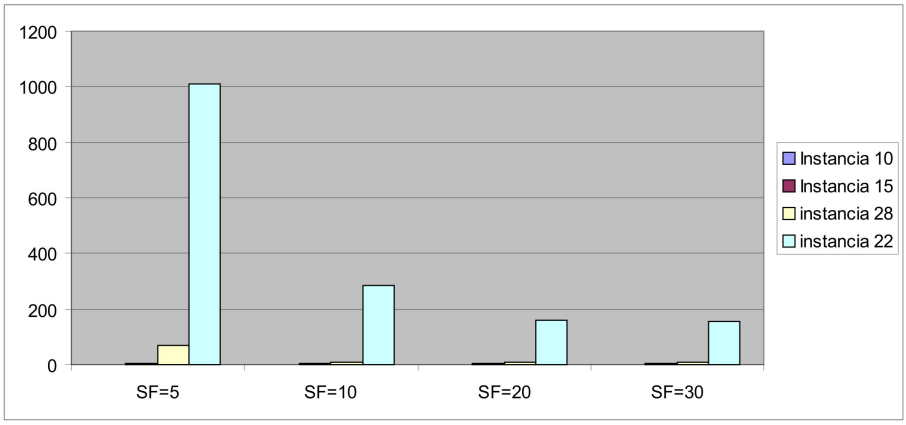

Figura 3.4: Tiempos de ejecución para algunas instancias en función del skip factor, para el algoritmo *branch and cut* con las desigualdades *vecinos*.

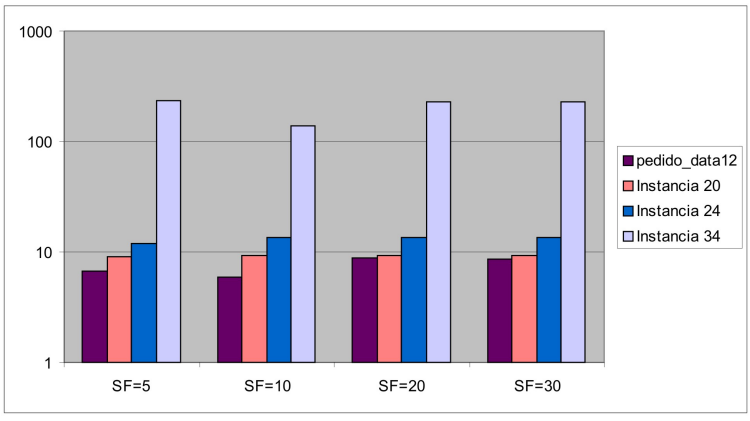

Figura 3.5: Tiempos de ejecución para algunas instancias en función del skip factor, para el algoritmo *branch and cut* con las desigualdades sobre los *nodos internos*.

Salvo para las familia de desigualdades (3.5) y (3.6), los algoritmos de separación involucran la construcción de un grafo intermedio con valores de e^x_{ij} y e^y_{ij} fraccionarios. Recordemos que este procedimiento construye un grafo a partir de la solución fraccionaria, y que esta construcción recibe un parámetro k que determina una cota inferior a los valores de las variables (asociadas con los ejes) que serán usadas en el grafo. Este factor es muy importante porque en función de este parámetro se podrán encontrar más o menos cortes. Por esta razón se realizaron varios experimentos con diferentes valores de k.

3.3.1 Desigualdades Garra

En el Cuadro 3.1 se encuentran los resultados de los experimentos realizados para la familia de desigualdades *garra* (3.1 y 3.2) para diferentes valores de k. Los tiempos de ejecución para la familia de desigualdades (3.1) empeoran sin importar los valores del parámetro utilizado. Notar también que la cantidad de cortes disminuye para valores de k más grandes, debido a que el grafo que utilizamos para buscar garras es muy poco denso. Sin embargo, cabe señalar que en los pocos casos en que se agregaron cortes de este tipo resultaron más eficaces que los aplicados para valores de k más chicos.

En este mismo cuadro también se pueden ver los cortes aplicados para la familia de desigualdades (3.2), notar que si bien los cortes aplicados mejoran un poco el tiempo, considerando sólo los resultados de los cortes (3.1) no son suficientes para mejorar los resultados del problema original. Aún en casos en donde la cantidad de cortes es alta el tiempo no disminuye sino que –al contrario– aumenta notablemente. La eficacia del corte es tan pobre que no alcanza a compensar el tiempo que se utiliza para buscar el corte y aplicarlo.

3.3.2 Desigualdades Ciclo

El Cuadro 3.2 contiene los resultados de haber aplicado la familia de desigualdades *ciclo* (3.3) para diferentes valores del parámetro k. Para esta familia de desigualdades los cortes agregados no son eficaces, y existen muy pocas instancias que se ven beneficiadas. Es interesante notar que la cantidad de cortes agregados aumenta cuando aumenta el valor k, esto puede ser debido a que cuando k es pequeño los valores fraccionarios de las variables "admitidas" son tan bajos que no alcanzan para que las desigualdades resultantes sean violadas, y por lo tanto las desigualdades generadas por el algoritmo son ignoradas y no se agregan como cortes.

3.3.3 Desigualdades Vecinos

Para esta familia de desigualdades el promedio de ejecución para los diferentes valores del parámetro k empeoró y en particular se pueden ver varias instancias cuyos tiempos de resolución se duplicaron. El Cuadro 3.3 muestra estos datos. Alrededor de 1370 cortes son aplicados cuando $k = 0{,}9$, sin embargo –al igual que en los casos anteriores– los cortes aplicados no son efectivos y en la mayoría de los casos afectan en forma negativa a los tiempos de ejecución.

3.3.4 Desigualdades para los nodos internos y externos

En el Cuadro 3.4 se pueden ver los resultados para las familias de desigualdades (3.5) y (3.6). Para el caso de la familia (3.5), el promedio de tiempo de ejecución disminuyó para los valores propuestos del parámetro k y muchas instancias se vieron beneficiadas, como por ejemplo la instancia 29 (que decrementó el tiempo de 3300 segundos a casi 400 segundos). Sin embargo,

$e^x_{ij} + e^y_{ij} + e^x_{il} + e^y_{il} + e^x_{ik} + e^y_{ik} + e^x_{jz} + e^x_{lg} + e^x_{km} + e^y_{jz} + e^y_{lg} + e^y_{km} \leq 5$								$y_j + y_n + y_l \geq e^x_{ij} + e^y_{ij} + e^x_{in} + e^y_{in} + e^x_{il} + e^y_{il} - 2$	
		k=0		k=0.75		k=0.9		k=0	
Instancia	T Sin Corte	Tiempo	Cortes	Tiempo	Cortes	Tiempo	Cortes	Tiempo	Cortes
Instancia 1	0.016	0.016	0	0.015	0	0.016	0	0.015	0
Instancia 2	0.016	0	0	0.016	0	0.016	0	0.015	0
Instancia 3	0.016	0.016	0	0	0	0	0	0.015	0
Instancia 4	0.031	0.031	0	0.031	0	0.016	0	0.031	0
Instancia 5	0.047	0.046	0	0.047	0	0.046	0	0.062	0
Instancia 6	0.031	0.031	0	0.015	0	0.031	0	0.031	0
Instancia 7	0.062	0.063	0	0.094	0	0.078	0	0.078	0
Instancia 8	0.156	0.218	0	0.218	0	0.219	0	0.234	0
Instancia 9	0.063	0.062	0	0.047	0	0.062	0	0.062	0
Instancia 10	0.187	0.218	0	0.218	0	0.234	0	0.234	0
Instancia 11	0.562	0.671	0	0.608	0	0.624	0	0.671	0
Instancia 12	5.132	5.507	1	7.66	0	7.629	0	8.378	0
Instancia 13	5.429	6.989	0	6.583	0	6.52	0	6.786	0
Instancia 14	2.481	3.525	1	3.447	2	3.291	0	3.229	0
Instancia 15	17.269	23.12	13	39.702	3	38.345	0	46.364	0
Instancia 16	1.216	1.248	2	1.513	0	1.513	0	1.607	0
Instancia 17	16.911	18.876	3	19.36	0	19.016	0	19.313	0
Instancia 18	13.275	26.115	4	20.842	0	20.468	0	20.779	0
Instancia 19	49.39	55.24	5	84.63	0	84.631	0	97.703	0
Instancia 20	9.844	57.19	35	42.51	6	40.42	1	51.387	0
Instancia 21	83.71	148.965	4	124.317	0	123.662	0	136.656	0
Instancia 22	3.198	4.758	4	4.352	0	4.01	0	4.306	0
Instancia 23	5.429	6.567	0	6.833	0	6.427	0	6.006	0
Instancia 24	14.711	21.341	0	20.405	0	18.86	0	20.249	1
Instancia 25	172.366	217.107	7	306.245	0	305.7	0	373.638	1
Instancia 26	13.572	19.204	3	16.24	0	16.271	0	17.519	0
Instancia 27	148.31	210.804	5	177.857	1	176.187	1	267.667	0
Instancia 28	516.067	2046.125	43	3600.05	15	1752.484	4	1432.261	4
Instancia 29	3312.666	1619.508	87	3605.978	21	3479.743	3	1711.503	4
Instancia 30	194.049	244.266	2	224.298	1	230.554	1	285.934	0
Instancia 31	236.186	471.201	22	2842.526	30	495.74	5	685.749	0
Instancia 32	46.192	58.657	0	55.131	0	54.584	0	57.471	0
Instancia 33	497.69	793.297	30	1087.171	6	1083.411	6	1465.442	0
Instancia 34	287.307	324.388	16	397.803	2	413.06	1	1585.064	1
Promedio | Suma	**166.28**	**187.80**	**287**	**373.43**	**87**	**246.58**	**22**	**244.30**	**0.32**

Cuadro 3.1: Desigualdades Garra

$\sum_{ij \in C} e^x_{ij} \leq \lvert C \rvert - 1$							
		k=0		k=0.75		k=0.9	
Instancia	T Sin Corte	Tiempo	Cortes	Tiempo	Cortes	Tiempo	Cortes
Instancia 1	0.016	0.016	0	0.016	0	0.016	0
Instancia 2	0.016	0.015	0	0.015	0	0.016	0
Instancia 3	0.016	0.016	0	0.016	0	0	0
Instancia 4	0.031	0.032	0	0.016	0	0.031	0
Instancia 5	0.047	0.078	0	0.047	0	0.047	0
Instancia 6	0.031	0.015	0	0.016	0	0.015	0
Instancia 7	0.062	0.078	0	0.078	0	0.078	0
Instancia 8	0.156	0.234	0	0.234	0	0.203	0
Instancia 9	0.063	0.062	0	0.062	0	0.047	0
Instancia 10	0.187	0.25	0	0.234	0	0.203	0
Instancia 11	0.562	0.639	0	0.64	0	0.593	0
Instancia 12	5.132	7.644	2	7.473	2	7.13	2
Instancia 13	5.429	6.583	0	6.537	0	6.224	0
Instancia 14	2.481	3.292	0	3.136	0	3.089	0
Instancia 15	17.269	37.191	4	30.03	3	41.761	12
Instancia 16	1.216	1.529	0	1.31	1	1.326	1
Instancia 17	16.911	16.208	3	11.559	2	11.123	2
Instancia 18	13.275	21.154	0	20.67	0	19.718	0
Instancia 19	49.39	76.472	1	74.428	2	72.322	2
Instancia 20	9.844	26.426	6	26.91	5	26.676	8
Instancia 21	83.71	126.517	1	121.899	0	117.063	0
Instancia 22	3.198	4.197	0	4.306	0	4.322	0
Instancia 23	5.429	6.567	0	6.786	0	6.662	0
Instancia 24	14.711	24.789	2	20.217	0	18.83	1
Instancia 25	172.366	324.014	2	298.243	0	292.986	0
Instancia 26	13.572	17.114	0	16.084	0	15.944	0
Instancia 27	148.31	230.319	3	233.315	1	200.228	2
Instancia 28	516.067	1162.722	2	1121.741	2	2364.773	17
Instancia 29	3312.666	3603.42	7	1561.024	16	3604.497	28
Instancia 30	194.049	329.381	1	304.576	1	271.473	2
Instancia 31	236.186	579.918	1	517.658	3	572.462	3
Instancia 32	46.192	59.686	0	54.632	0	53.82	0
Instancia 33	497.69	1572.272	2	1132.146	0	1138.152	0
Instancia 34	287.307	556.253	3	425.337	1	421.28	1
Promedio | Suma	**166.28**	**258.6795**	**40**	**176.51**	**39**	**272.73**	**81**

Cuadro 3.2: Desigualdades Ciclo

$\sum e^x_{ij} + e^y_{ij} \geq 2 \times (\sum x_i - \lvert c \rvert + 1)$							
		k=0		k=0.5		k=0.9	
Instancia	T Sin Corte	Tiempo	Cortes	Tiempo	Cortes	Tiempo	Cortes
Instancia 1	0.016	0.016	0	0.015	0	0.015	0
Instancia 2	0.016	0.016	0	0.016	0	0.016	0
Instancia 3	0.016	0	0	0.015	0	0.016	0
Instancia 4	0.031	0.031	1	0.032	1	0.031	1
Instancia 5	0.047	0.047	1	0.047	1	0.047	1
Instancia 6	0.031	0.015	1	0.032	1	0.016	1
Instancia 7	0.062	0.078	1	0.062	1	0.078	1
Instancia 8	0.156	0.218	2	0.171	1	0.234	2
Instancia 9	0.063	0.062	0	0.047	0	0.062	0
Instancia 10	0.187	0.234	2	0.374	4	0.265	3
Instancia 11	0.562	0.484	2	1.045	4	0.39	1
Instancia 12	5.132	6.459	6	11.544	17	8.174	12
Instancia 13	5.429	8.175	3	12.979	4	8.845	5
Instancia 14	2.481	3.541	3	2.652	3	2.699	3
Instancia 15	17.269	30.389	43	22.401	27	16.833	30
Instancia 16	1.216	1.061	3	2.324	11	1.139	3
Instancia 17	16.911	6.895	5	5.756	3	4.025	1
Instancia 18	13.275	27.706	7	28.533	9	17.738	4
Instancia 19	49.39	139.028	35	59.826	12	88.375	18
Instancia 20	9.844	113.943	245	50.294	113	20.327	29
Instancia 21	83.71	286.324	35	169.058	12	199.401	17
Instancia 22	3.198	3.932	6	2.652	2	2.465	2
Instancia 23	5.429	5.195	1	6.52	1	8.892	2
Instancia 24	14.711	17.956	3	20.421	7	21.419	5
Instancia 25	172.366	254.921	25	206.233	23	153.755	20
Instancia 26	13.572	13.65	11	7.535	3	9.111	3
Instancia 27	148.31	199.806	7	609.215	31	224.642	8
Instancia 28	516.067	1881.482	88	2716.788	185	1795.743	95
Instancia 29	3312.666	3603.389	489	819.535	86	3600.363	473
Instancia 30	194.049	207.903	6	273.314	13	671.709	43
Instancia 31	236.186	430.219	39	322.61	22	877.989	96
Instancia 32	46.192	38.766	2	85.005	7	81.089	5
Instancia 33	497.69	571.79	40	862.264	60	3355.488	408
Instancia 34	287.307	347.289	37	172.693	20	530.044	95
Promedio | Suma	**166.28**	**241.20**	**1149**	**190.36**	**684**	**344.15**	**1387**

Cuadro 3.3: Desigualdades Vecinos

		$\sum_{i,j \in P} e^x_{ij} \geq x_i$		$\sum_{i,j \in P} e^y_{ij} \geq y_i$	
Instancia	T Sin Corte	Tiempo	Cortes	Tiempo	Cortes
Instancia 1	0.016	0.031	0	0.032	0
Instancia 2	0.016	0.015	0	0.016	0
Instancia 3	0.016	0.015	0	0.016	0
Instancia 4	0.031	0.031	4	0.032	0
Instancia 5	0.047	0.062	2	0.047	0
Instancia 6	0.031	0.015	3	0.031	0
Instancia 7	0.062	0.063	2	0.078	0
Instancia 8	0.156	0.172	4	0.172	0
Instancia 9	0.063	0.047	0	0.046	0
Instancia 10	0.187	0.39	6	0.218	0
Instancia 11	0.562	0.609	6	0.624	0
Instancia 12	5.132	8.378	18	5.461	1
Instancia 13	5.429	8.112	12	5.928	0
Instancia 14	2.481	1.389	7	2.605	0
Instancia 15	17.269	12.106	38	17.035	25
Instancia 16	1.216	0.998	4	1.404	1
Instancia 17	16.911	5.974	12	17.581	0
Instancia 18	13.275	13.728	9	13.322	5
Instancia 19	49.39	55.365	11	44.429	13
Instancia 20	9.844	8.877	28	9.86	11
Instancia 21	83.71	190.571	16	88.687	3
Instancia 22	3.198	2.652	1	3.916	0
Instancia 23	5.429	4.617	10	6.053	0
Instancia 24	14.711	12.605	12	16.13	0
Instancia 25	172.366	166.734	25	171.508	3
Instancia 26	13.572	8.174	11	13.619	0
Instancia 27	148.31	131.758	14	159.542	2
Instancia 28	516.067	682.098	79	659.806	23
Instancia 29	3312.666	390.548	315	297.151	239
Instancia 30	194.049	184.814	10	249.633	5
Instancia 31	236.186	486.77	37	321.159	82
Instancia 32	46.192	77.58	17	52.244	1
Instancia 33	497.69	300.926	46	339.505	29
Instancia 34	287.307	124.473	13	272.846	7
Promedio | Suma	**166.28**	**84.72**	**772**	**81.49**	**450**

Cuadro 3.4: Desigualdades Nodos

salvo el caso anterior, estas diferencias de tiempo son pequeñas y hay instancias que fueron afectadas negativamente por la generación de estos cortes. La situación es la misma para la familia (3.6): si bien el promedio del tiempo de ejecución promedio es menor, los resultados no son contundentes.

3.4 Conclusiones

Para todas las familias de desigualdades se probó con distintas frecuencias de ejecución de los algoritmos de separación y con diferentes valores del umbral k para crear ejes en el grafo intermedio, llegando a la conclusión de que ningún algoritmo de separación propuesto logra generar cortes que sean contundentes en relación con la performance del procedimiento global. Si bien hay algunas instancias que se ven beneficiadas, existen más intancias perjudicadas.

En la Figura 3.6 se muestra el grafo que se obtiene a partir de la solución (fraccionaria) de la relajación lineal para la instancia número 19. Los ejes pintados de color verde representan a las variables e^x y los ejes pintados con rojo representan a las variables e^y, también se muestran

los valores fraccionarios de las variables asociadas a los ejes. En este caso, la suma de cualquier combinación que forme una garra es menor a cinco, con lo cual, si bien existen garras en este grafo, la solución fraccionaria no viola ninguna desigualdad *garra* que, por lo tanto, no genera cortes. Se da una situación similar para el resto de las familias de desigualdades: los valores fraccionarios de las variables son tan pequeños que no alcanza para que las desigualdades sean violadas y aporten información de valor.

En un principio se pensó que los cortes agregados no eran lo suficientemente fuertes para disminuir significativamente la región factible. Sin embargo, durante las corridas y observando el comportamiento del algoritmo se notó que *la función objetivo del óptimo entero coincide, en casi todos los casos, con la función objetivo de la relajación lineal.* Es decir, la cota dual inicial (en el primer nodo del árbol de enumeración) y la función objetivo del óptimo entero (obtenido cuando el algoritmo *branch and bound* termina) son iguales o tienen un *gap* muy chico. Este efecto es poco usual y posiblemente es uno de los motivos que explica por qué los cortes antes mencionados no mejoran la cota dual y en consecuencia los tiempos no mejoran.

La Figura 3.7 muestra parte del log que arroja *CPLEX* para dicha instancia y el Cuadro 3.5 muestra para todas las instancias el *gap* en el primer nodo de la ejecución, las cotas primal y dual y la función objetivo de la solución óptima del problema. Se puede ver que en casi todas las instancias consideradas, la cota dual inicial dada por la función objetivo de la relajación lineal ya es el valor óptimo, pero tarda mucho tiempo en encontrar una solución factible cuya función objetivo coincida con este valor.

En función de estas observaciones, no parecería ser conveniente la búsqueda de nuevas familias de desigualdades o el diseño de algoritmos de separación más eficientes. Por el contrario, posiblemente una *heurística primal* que sea capaz de encontrar buenas soluciones factibles sea más útil en el contexto de un algoritmo *branch and bound*, dado que podría ayudar a *CPLEX* a encontrar soluciones factibles. El siguiente capítulo tiene como objetivo estudiar dicha problemática.

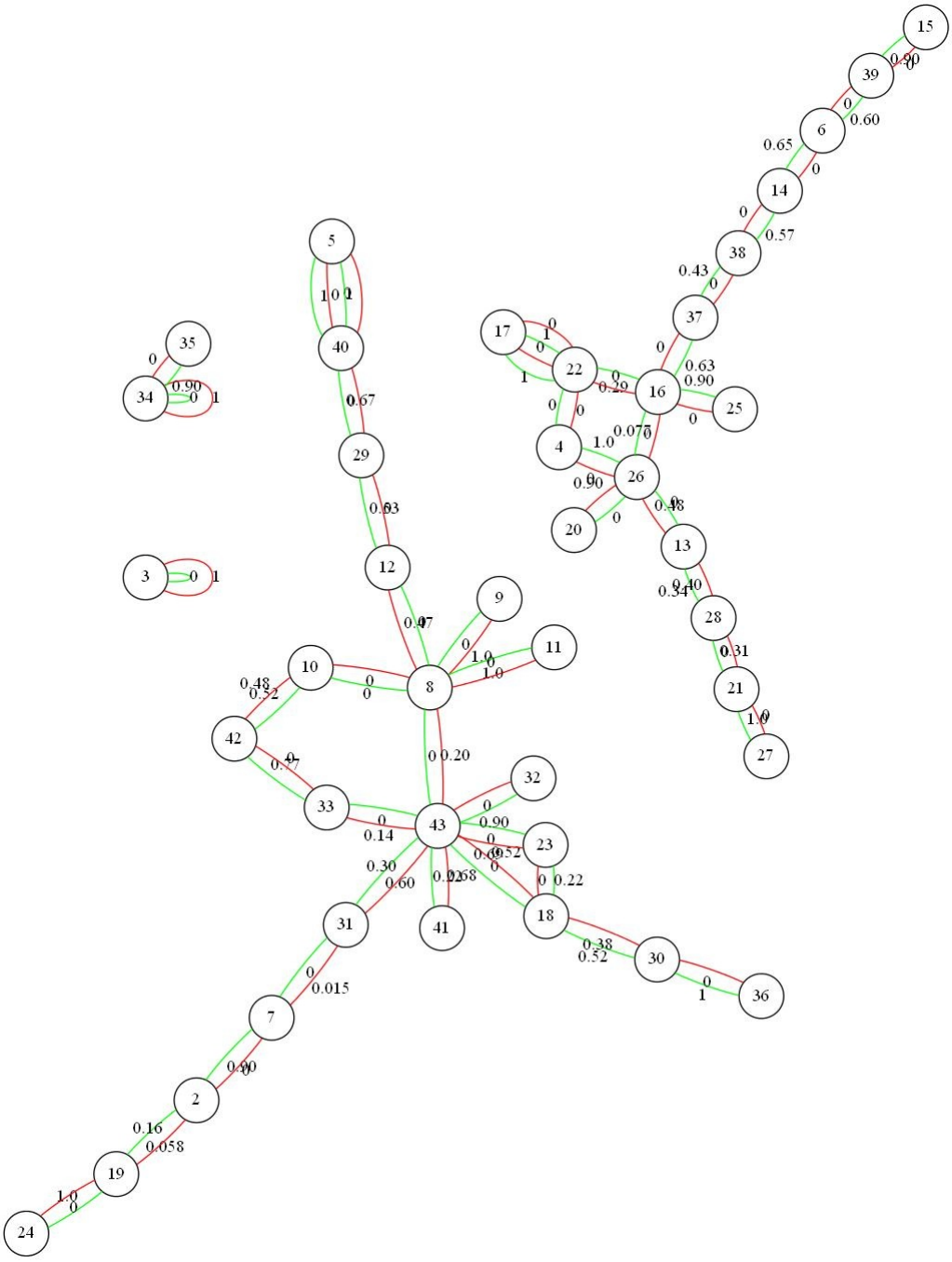

Figura 3.6: Grafo asociado con una solución fraccionaria hallada por el algoritmo *branch and cut* para la instancia 19.

```
      Nodes                                         Cuts/
 Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap         Variable B NodeID Parent  Depth

    0     0      768.1219    87                  768.1219      420
*     0+    0                       768.8417      768.1219      420    0.09%
      0     2      768.1219    28      768.8417      768.1219      420    0.09%                       0            0
Elapsed real time =    4.80 sec. (tree size =  0.03 MB, solutions = 0)
      3     5      768.1219    42      768.8417      768.1219      684    0.09%         ex#6#39 U      3       1       2
     10     9      768.1219    50      768.8417      768.1219      832    0.09%        ex#16#26 U     10       9       7
     21    18      768.1219    50      768.8417      768.1219     1125    0.09%        ex#29#31 U     21      20      17
     36    29      768.1219    40      768.8417      768.1219     1485    0.09%        ex#10#42 U     36      35      29
     54    40      768.1219     9      768.8417      768.1219     1888    0.09%        ex#32#42 U     54      53      41
     80    49      768.1219     9      768.8417      768.1219     2215    0.09%        ey#38#25 D     80      79      54
                                                     User: 2
    115    50      768.1219    13      768.8417      768.1219     2597    0.09%         ex#7#10 U    115     107      53
                                                     User: 8
    160    49      768.1219    10      768.8417      768.1219     2966    0.09%         ey#5#14 D    160     159      54
                                                     User: 5
    214    46      768.1219     8      768.8417      768.1219     3400    0.09%         ex#9#40 D    214     213      46
    438    51      768.1219    20      768.8417      768.1219     6068    0.09%        ex#20#22 U    438     437      45
                                                     User: 7
Elapsed real time =   23.96 sec. (tree size =  1.52 MB, solutions = 0)
    680    45      768.1219    15      768.8417      768.1219     8852    0.09%        ex#11#42 D    680     679      37
    912    68      768.1219    15      768.8417      768.1219    11487    0.09%         ey#4#25 D    912     911      41
                                                    User: 15
   1192    66      768.1219    29      768.8417      768.1219    14484    0.09%        ex#20#22 U   1192    1191      45
                                                    User: 10
   1487    57      768.1219    20      768.8417      768.1219    17313    0.09%        ey#35#40 U   1487    1486      40
   1756    73      768.2483    38      768.8417      768.1219    19928    0.09%         ex#4#17 D   1756    1755      44
   1969   108      768.1219    11      768.8417      768.1219    21806    0.09%         ey#3#40 D   1969    1968      58
   2153   108      768.1219    18      768.8417      768.1219    23253    0.09%            y#33 D   2153    2140      45

    .      .          .        .         .             .         .       .              .          .      .       .
    .      .          .        .         .             .         .       .              .          .      .       .
    .      .          .        .         .             .         .       .              .          .      .       .

Nodefile size = 36.26 MB (8.38 MB after compression)
 1258219 41787      768.1219    22      768.6486      768.1219 12953604    0.07%         ey#32#12 D 1258219 1258218     71
 1263412 41761    infeasible            768.6486      768.1219 13018707    0.07%         ey#31#40 U 1263412 1263407     77
*1265800+41663                          768.5861      768.1219 13046622    0.06%
                                                     User: 1
*1265800+40164                          768.2483      768.1219 13046622    0.02%
*1267000+     0                         768.1219      768.1219 13061530    0.00%
                                                     User: 2
```

Figura 3.7: Log de *CPLEX* para la intancia 19.

Instancia	Primer Solución Entera	Primer Relajación lineal	Funcion Objetivo Óptimo	gap
Instancia 1	204.5552	204.5552	204.5552	0.00 %
Instancia 2	138.4739	138.4739	138.4739	0.00 %
Instancia 3	108.1802	108.1802	108.1802	0.00 %
Instancia 4	199.3205	199.3205	199.3205	0.00 %
Instancia 5	116.2544	116.2544	116.2544	0.00 %
Instancia 6	232.9237	232.9237	232.9237	0.00 %
Instancia 7	97.5872	59.5361	65.8779	38.99 %
Instancia 8	39.7904	39.7638	39.76383	0.07 %
Instancia 9	665.1293	665.1293	665.1293	0.00 %
Instancia 10	589.8818	589.8818	589.8818	0.00 %
Instancia 11	140.052	138.9282	138.9282	0.80 %
Instancia 12	279.7578	279.7578	279.7578	0.00 %
Instancia 13	932.8774	881.5306	881.5306	5.50 %
Instancia 14	2132.2396	2132.2396	2132.2396	0.00 %
Instancia 15	653.4546	622.6999	622.7005	4.71 %
Instancia 16	966.903	680.8923	680.8922	29.58 %
Instancia 17	743.0591	707.0906	707.0906	4.84 %
Instancia 18	574.077	557.7429	557.7429	2.85 %
Instancia 19	768.8417	768.1219	768.1218	0.09 %
Instancia 20	293.5649	283.962	284.3712	3.27 %
Instancia 21	350.9971	343.0665	343.0836	2.26 %
Instancia 22	423.7602	404.3016	404.3016	4.59 %
Instancia 23	648.8221	615.3488	615.3488	5.16 %
Instancia 24	483.6359	480.614	480.6140	0.62 %
Instancia 25	1033.7158	1030.67	1030.6670	0.29 %
Instancia 26	2007.28	1624.1213	1624.1212	19.09 %
Instancia 27	2041.3379	1923.8498	1923.8735	5.76 %
Instancia 28	1090.1537	1023.920032	1023.9200	6.08 %
Instancia 29	761.9839	706.9068	706.9534	7.23 %
Instancia 30	1279.0403	1258.6005	1258.6005	1.60 %
Instancia 31	1489.3375	1404.7088	1404.7088	5.68 %
Instancia 32	1087.0284	1078.022	1078.0220	0.83 %
Instancia 33	2474.2111	2180.611	2180.8072	11.87 %
Instancia 34	818.1024	739.6355	739.6355	9.59 %

Cuadro 3.5: Comparación entre la primer solución entera encontrada, la primer relajacion lineal y la *función objetivo* de la solución óptima

Capítulo 4

Heurística primal

Como se comentó en la sección anterior aplicar planos de corte durante el algoritmo *branch and bound* no mejora la cota dual porque esta cota comienza siempre con una buena aproximación al valor óptimo. En consecuencia, la mayor parte del tiempo consumido por *CPLEX* se destina a buscar una solución factible cuya función objetivo coincida (o esté cerca) de la cota dual dada por la relajación lineal. En los experimentos realizados, se observó que muchas soluciones fraccionarias generadas a lo largo del algoritmo tienen una función objetivo que coincide con la función objetivo del óptimo entero. Por estos motivos, se propone en este capítulo una heurística primal que intenta construir soluciones factibles a partir de las soluciones fraccionarias halladas a lo largo de la ejecución. La intención es que esta heurística pueda ayudar al algoritmo a mejorar las cotas primales, y así cerrar el gap lo más temprano posible.

4.1 Desarrollo

A partir de una solución fraccionaria arrojada por *CPLEX*, la heurística fija algunas variables e^x y e^y para dejar prefijada parte de la estructura de la solución. Se deja libre el resto de las variables (y, en particular, las variables m que representan los metros lineales de cada combinación), y se resuelve un modelo de programación entera restringido, igual al modelo original pero con la salvedad de que se fijan los valores de algunas variables. Se utiliza el mismo paquete *CPLEX* para su resolución. La intención es que *CPLEX* pueda aprovechar la estructura prefijada de la solución y utilizar las variables libres para armar una solución factible (idealmente óptima para el subproblema) y, de esta forma, ayudar al procedimiento global con la cota primal hallada.

Se puede separar la heurística en dos fases. La primera fase consiste en construir el grafo asociado a la solución fraccionaria. Los nodos de este grafo corresponden a los pedidos, y los ejes están dados por todos los pares de pedidos ij cuyos valores de las variables m_{ijc} sean mayores a 0 para algún c. Notar que el procedimiento cambia con respecto a la construcción de los algoritmos de separación planteados en el capítulo anterior, dado que en ese caso interesa que el grafo resultado contenga todos los ejes asociados con las variables m_{ijc} que tengan valores no nulos. Una vez construido este grafo, se quitan en forma golosa los ejes que generen *garras* y *ciclos*, de esta forma se obtiene un grafo *caterpillar* que servirá de input para la segunda fase.

La segunda fase consiste en crear un nuevo problema de programación entera utilizando el modelo inicial del problema original y fijando las variables e^x_{ij} y e^y_{ij} en 1 de acuerdo con el

grafo obtenido en la primera fase. Es decir, se fijan los valores $e^x_{ij} = e^y_{ij} = 1$ para todos los ejes ij presentes en el grafo construido en la primera fase. El resto de las variables se deja libre. La intención es aprovechar la estructura de la solución proporcionada por la solución fraccionaria, intentando aprovechar las "cadenas" de pedidos (correspondientes a caminos en este grafo) que se hayan formado. El problema auxiliar de optimización es ejecutado y, si *CPLEX* encuentra una solución factible mejor que la cota primal obtenida hasta el momento, se setea este valor como nueva cota primal del procedimiento global.

En el Algoritmo 1 se muestra el pseudocódigo de la heurística. La heurística se puede ejecutar luego de resolver la relajación lineal asociada a cada subproblema, siempre que dicha relajación lineal sea factible y su óptimo sea fraccionario. Con esta solución fraccionaria se construye el grafo compuesto por las variables e^x y e^y mayores a cero (línea 3). Debido a que las variables son fraccionarias, muy probablemente, el grafo encontrado contendrá *ciclos* y *garras*, es por esto que para cada componente conexa quitamos la mínima cantidad posible de ejes para que la componente conexa resulte *caterpillar* (líneas 5 y 6). Con esto termina la primera fase dando origen a un grafo que es unión de grafos *caterpillar*, y que no necesariamente puede dar origen a una solución factible. La segunda fase corresponde a crear un modelo de programación entera igual al modelo del problema original, pero fijando las variables e^x_{ij} y e^y_{ij} en 1 para todos los ejes ij del grafo obtenido en la fase anterior (línea 8). Notar que las variables m quedan libres. Si el problema auxiliar resulta factible se utiliza esta solución para actualizar la cota primal del procedimiento global. Los detalles de la implementación de los algoritmos secundarios se pueden consultar en el Anexo A.3.4.

```
Algorithm: main
begin
    grafo ← armarGrafo(cplex, xs, 0)
    for s ∈ grafo.getComponenteConexa() do
        quitarGarras(s)
        quitarCiclos(s)
    end
    nuevoCplex ← newcplex()
    for e eje de grafo do
        nuevoCplex ← agregar restriccion eje ex_ij == 1 y ey_ij == 1
    end
    if nuevoCplex es optimo then
        setear variables al cplex original
    end
end
```

Algorithm 1: Heurística

4.1.1 Ejemplo

Para mostrar un ejemplo tomamos una instancia de la muestra estudiada que tuviese una cantidad de pedidos relativamente pequeña pero que aprovechara la heurística al máximo. Es por esto que se seleccionó la instancia 12 con catorce pedidos de diferentes configuraciones y alrededor de 39000 planchas a producir. Para esta instancia se aplica sólo una vez la heurística, reduciendo el tiempo de optimización de cinco segundos a dos segundos.

La Figura 4.1 muestra el grafo intermedio antes de aplicar la heurística. Notar que este

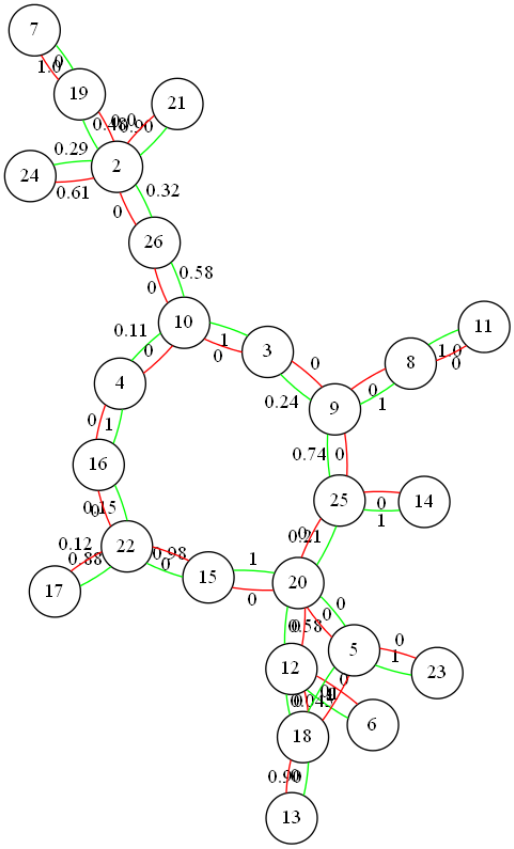

Figura 4.1: Grafo para la instancia 12 antes de aplicar la heurística. Los ejes verdes corresponden a las variables e^x y los ejes rojos corresponden a las variables e^y.

grafo dista bastante de ser un grafo *caterpillar*. Siguiendo con el algoritmo detallado anteriormente, en este punto, la heurística busca ejes para quitarlos del grafo y evitar ciclos y garras. En este caso puntual, los ejes que se quitan son: (15,20), (4,10), (5,20), (8,9) y (20,25). La Figura 4.2 muestra el grafo resultado de quitar dichos ejes. Un nuevo problema de optimización auxiliar es creado con las variables e^x y e^y de este grafo fijadas en 1, se optimiza y se obtiene una solución factible óptima con una función objetivo de 292,591. Esta solución tiene un *gap* de 4.39 % con respecto al problema original.

Después de agregar la solución encontrada al problema original se obtiene como solución final el grafo de la Figura 4.3. En esta solución se agregaron los ejes (8,9) y (20,15) los cuales habían sido removidos por la heurística, se podría pensar que si los ejes (15,20), (4,10) y (5,20) hubieran sidos quitados en primera intancia habríamos llegado a la solución óptima más rápidamente y la solución del problema de optimización intermedio sería la misma que el problema original. Una posible solución es que la heurística extraiga los ejes con valores de las variables e^x y e^y más pequeños. En la Figura 4.1, el eje (8,9) tiene un valor de 0.9 y el eje (20,25) tiene un valor de 1.

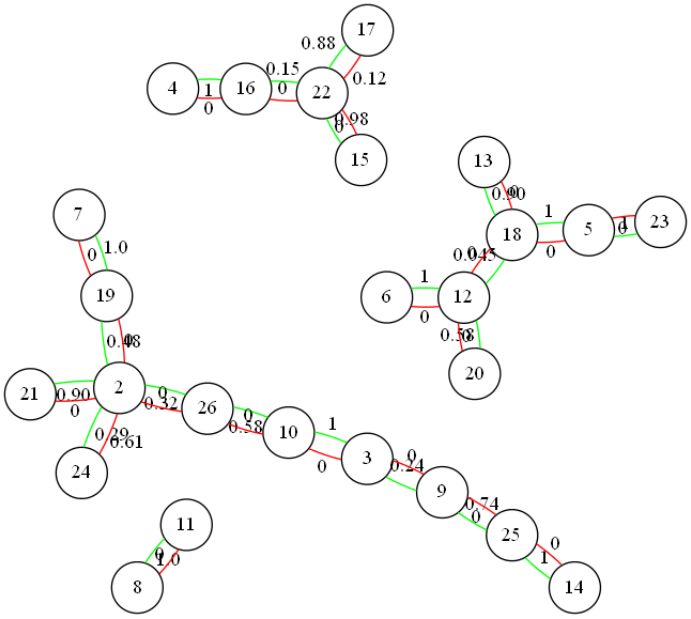

Figura 4.2: Grafo para la instancia 12 despues de aplicar la heurística (sin *ciclos* ni *garras*).

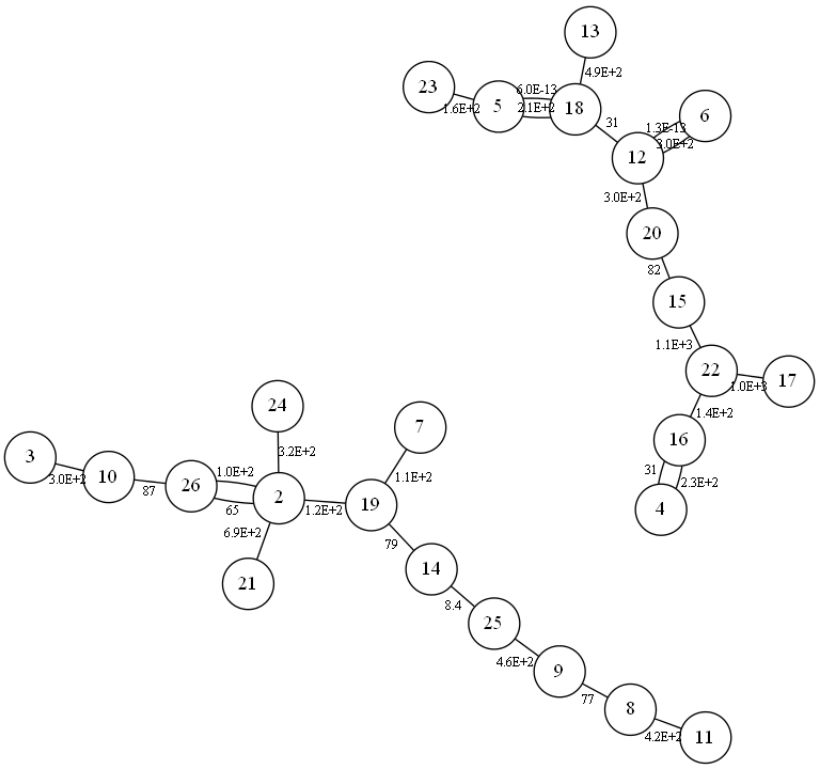

Figura 4.3: Grafo asociado a la solución óptima de la instancia 13.

4.2 Experiencia computacional

Se realizaron experimentos fortaleciendo el algoritmo *branch and bound* con la ejecución de la heurística en todos o algunos de los subproblemas del árbol de enumeración. Al aplicar la heurística hay un incremento de tiempo importante debido a que es costoso para *CPLEX* en cada iteración crear un nuevo problema y resolverlo, ademas del tiempo extra utilizado para armar el grafo y quitar las garras y los ciclos. Para evitar este incremento y lograr algún balance entre la complejidad de la heurística y su eficacia se estudiaron las siguientes estrategias:

1. Aplicar la heurística en todos los nodos.
2. Aplicar la heurística en todos los nodos, pero con una tolerancia de gap del 10 % sobre el procedimiento global.
3. Aplicar sólo una vez la heurística (en la relajación lineal).
4. Aplicar la heurística en todos los nodos, pero con un tiempo límite al problema auxiliar de 30 segundos.
5. Aplicar la heurísitica cada 100 nodos.

En el Cuadro 4.1 se muestran los tiempos de ejecución de *CPLEX* sin la heurística (columna "Sin heurística") y con las estrategias planteadas. Es interesante observar que no resulta práctico ejecutar la heurística en todos los nodos sin límites extra de tiempo o de gap, dado que la gran cantidad de instancias de *CPLEX* abiertas hace que la máquina virtual de java tenga problemas serios de performance. Dichos problemas se observan para las instancias cuyos tiempos de optimización se encuentras subrayados. En estos casos la heurística es aplicada cada 30 nodos, de modo de reducir la cantidad de problemas a resolver y se agrega la columna "Nodo con problemas" en la tabla Tabla 4.2 para estas instancias puntuales donde se muestra la cantidad de veces que se aplicó la heurística antes de que la máquina virtual tuviera problemas. Esta tabla muestra en detalle la ejecución donde se hace un promedio de la función objetivo, el tiempo promedio de todos los problemas auxiliares y la función objetivo del problema auxiliar más cercano a la función objetivo final para cada instancia. Notar que el promedio de duración del tiempo de optimización del problema auxiliar (columna "Promedio tiempo auxiliar") es pequeño en comparación al tiempo total de resolución para el problema original y el promedio de la función objetivo (columna "Promedio función objetivo auxiliar") es bastante parecido a la función objetivo de la solución final (columna "Función objetivo óptimo").

Teniendo en cuenta que el promedio del tiempo de ejecución de los problemas auxiliares no supera los 30 segundos, la estrategia 4 es similar a correr la heuristica en todos los nodos (y, de hecho, el problema con la máquina virtual de java se repite en este caso). Se observa un incremento en los tiempos de ejecución para las instancias que superan los 30 segundos. Se puede pensar que el tiempo propuesto para la resolución del problema auxiliar es bajo, y por lo tanto el problema auxiliar nunca llega a una solución óptima y la solución es descartada. Por dicho motivo, se descartó la opción de reducir aún más el límite de tiempo para el problema auxiliar.

De todos los criterios probados, aplicar la heurística en todos los nodos con una tolerancia de *gap* del 10 % para el procedimiento global (estrategia 2) fue el que mejores resultados

Instancia	Sin heurística	Con heurística (1)	$Gap \leq 10\,\%$(2)	Una vez (3)	Máx. 30 seg. (4)	100 nodos(5)
Instancia 1	0.016	0.031	0.031	0.031	0.031	0.047
Instancia 2	0.016	0.031	0.031	0.031	0.031	0.031
Instancia 3	0.016	0.031	0.016	0.031	0.015	0.031
Instancia 4	0.031	0.031	0.031	0.031	0.031	0.031
Instancia 5	0.047	0.031	0.062	0.047	0.047	0.047
Instancia 6	0.031	0.016	0.031	0.031	0.031	0.031
Instancia 7	0.062	0.452	0.312	0.141	0.452	0.125
Instancia 8	0.156	0.125	0.125	0.171	0.14	0.187
Instancia 9	0.063	0.078	0.062	0.063	0.078	0.063
Instancia 10	0.187	0.094	0.125	0.078	0.078	0.062
Instancia 11	0.562	0.234	0.202	0.421	0.25	0.452
Instancia 12	5.132	1.139	1.17	1.139	1.123	1.154
Instancia 13	5.429	3.073	1.216	5.772	2.137	3.276
Instancia 14	2.481	0.515	0.53	0.531	0.53	0.578
Instancia 15	17.269	30.561	0.858	14.664	21.949	14.945
Instancia 16	1.216	0.873	0.624	0.967	0.843	0.936
Instancia 17	16.911	8.034	1.217	6.506	7.862	6.599
Instancia 18	13.275	13.073	3.276	11.529	13.182	11.762
Instancia 19	49.39	45.162	4.291	3246.787	46.832	3244.976
Instancia 20	9.844	160.806	0.858	9.516	197.31	9.469
Instancia 21	83.71	40.467	14.414	41.746	35.241	40.826
Instancia 22	3.198	2.699	0.437	9.438	2.621	9.407
Instancia 23	5.429	4.181	2.309	4.353	4.274	4.134
Instancia 24	14.711	5.865	1.997	4.088	6.162	4.181
Instancia 25	172.366	13.323	8.767	13.79	13.869	13.26
Instancia 26	13.572	9.906	2.449	10.374	12.745	10.14
Instancia 27	148.31	171.382	37.846	154.145	223.487	143.177
Instancia 28	516.067	943.697	46.036	875.711	843.263	861.936
Instancia 29	3312.666	2952.927	14.524	251.458	3634.683	250.819
Instancia 30	194.049	374.605	34.336	73.773	263.486	72.65
Instancia 31	236.186	351.626	37.019	247.261	657.56	434.166
Instancia 32	46.192	59.499	7.254	36.675	87.642	87.501
Instancia 33	497.69	432.497	42.963	855.042	387.335	920.266
Instancia 34	287.307	392.936	9.235	109.653	340.223	116.72
Promedio	**166.28**	**177.05**	**8.07**	**176.05**	**200.16**	**184.23**

Cuadro 4.1: Heurística aplicada para diferentes criterios de parada.

Instancia	Tiempo Optimización	Función Objetivo óptimo	#Heur. Aplicadas	Nodo con problemas	Promedio Función Obj. auxiliar	Promedio tiempo auxiliar	Mejor Fun. Obj. Auxiliar
Instancia 1	0.031	204.5551667	1	-	204.5551667	0.015	204.5551667
Instancia 2	0.031	138.4739	1	-	138.4739	0.016	138.4739
Instancia 3	0.031	108.1801733	1	-	108.1801733	0.016	108.1801733
Instancia 4	0.031	199.320516	1	-	199.320516	0.016	199.320516
Instancia 5	0.031	116.2544	1	-	116.2544	0.015	116.2544
Instancia 6	0.016	232.923678	1	-	232.923678	0.000	232.923678
Instancia 7	0.452	65.877913	16	-	102.1738922	0.011	65.877913
Instancia 8	0.125	39.76383333	1	-	39.79043333	0.031	39.76383333
Instancia 9	0.078	665.12933	1	-	665.12933	0.031	665.12933
Instancia 10	0.094	589.8818205	1	-	589.8818205	0.031	589.8818205
Instancia 11	0.234	138.9281724	2	-	139.4900974	0.031	138.9281724
Instancia 12	1.139	279.757824	1	-	279.757824	0.609	292.5918469
Instancia 13	3.073	881.5306385	1	-	932.8077215	0.998	881.5306385
Instancia 14	0.515	2132.239622	1	-	2132.239622	0.218	2132.239622
Instancia 15	30.561	622.7055067	16	-	737.4665539	1.021	622.7486665
Instancia 16	0.873	680.8922983	5	-	766.9812583	0.053	680.8922983
Instancia 17	8.034	707.0905877	7	120	833.8567777	0.737	707.0905877
Instancia 18	13.073	557.7429407	4	-	584.0364906	2.195	557.7429407
Instancia 19	45.162	768.12188	13	-	787.1664086	2.137	769.826576
Instancia 20	160.806	284.3712	120	-	286.0191761	0.801	284.3712
Instancia 21	40.467	343.0664626	3	98	346.370285	6.230	343.846188
Instancia 22	2.699	404.301613	7	-	408.8958797	0.147	405.3092938
Instancia 23	4.181	615.3488435	1	-	648.8221135	1.466	615.3488435
Instancia 24	5.865	480.6140413	3	-	482.628632	0.925	483.6359273
Instancia 25	13.323	1030.710448	1	-	1033.715832	4.103	1030.801568
Instancia 26	9.906	1624.121255	1	-	1624.121255	2.777	1624.121255
Instancia 27	171.382	1923.849756	2	-	1944.106588	27.776	1945.232233
Instancia 28	943.697	1024.014628	15	7	1106.76483	32.402	1023.952013
Instancia 29	2952.927	706.9130819	78	30	765.2504877	28.589	706.96929
Instancia 30	374.605	1258.600529	2	82	1261.138416	59.061	1258.600529
Instancia 31	351.626	1404.70884	6	72	1494.122684	26.468	1404.70884
Instancia 32	59.499	1078.021976	1	14	1078.021976	12.574	1078.608851
Instancia 33	432.497	2180.80719	9	102	2232.527628	16.938	2182.370378
Instancia 34	392.936	739.6916158	14	111	1070.532945	14.232	739.7059646

Cuadro 4.2: Detalle de las heurísticas aplicadas en todos los nodos.

obtuvo. Esto es bastante intuitivo si se tiene en cuenta el Cuadro 3.5 donde se ve que en la mayoría de las instancias durante la ejecución del primer subproblema el *gap* no supera el 10 %. Si bien esto hace que la solución final puede no ser óptima, en la mayoría de las instancias el *gap* real es inferior al 6 % y los beneficios en cuanto al tiempo de ejecución son contundentes. El promedio de tiempos de ejecución bajó de 166 segundos a sólo 8 segundos e instancias para las cuales se tenían tiempos de ejecución cercanos a una hora pasaron a tardar unos pocos segundos.

Para analizar cuánto de esta ganancia está generada por la introducción de la heurística, en el el Cuadro 4.3 se comparan la ejecución de *CPLEX* con parámetros por defecto, la ejecución de *CPLEX* con una tolerancia para el *gap* de 10 %, y la ejecución de *CPLEX* utilizando la heurística en todos los subproblemas y con una tolerancia para el *gap* de 10 % para el problema global (correspondiente a la estrategia 2). El agregado de la heurística al considerar un *gap* máximo de 10 % logra un promedio de tiempos de ejecución de 8,07 segundos con relación a los 31,28 segundos promedio medidos cuando no se utiliza la heurística. Además, las soluciones obtenidas se encuentran a una distancia de 2.44 % de las soluciones óptimas, mejorando por sobre el 3.95 % medido cuando no se utiliza la heurística.

Optimización sin Heurística				Optimización sin Heurística con $Gap = 10\,\%$			Optimización con Heurística (Gap 10 %)		
Instancia	Funcion Objetivo Óptimo	Gap Relajación (%)	Tiempo Opt.	Funcion Objetivo	Gap Real(%)	Tiempo Opt.	Funcion Objetivo Óptima	Gap Real(%)	Tiempo Opt.
Instancia 1	204.5551667	0.00	0.016	204.5551667	0.00	0.015	204.5551667	0.00	0.031
Instancia 2	138.4739	0.00	0.016	138.4739	0.00	0.016	138.4739	0.00	0.031
Instancia 3	108.1801733	0.00	0.016	108.1801733	0.00	0.015	108.1801733	0.00	0.016
Instancia 4	199.320516	0.00	0.031	199.320516	0.00	0.016	199.320516	0.00	0.031
Instancia 5	116.2544	0.00	0.047	118.91384	2.29	0.032	116.2544	0.00	0.062
Instancia 6	232.923678	0.00	0.031	232.923678	0.00	0.031	232.923678	0.00	0.031
Instancia 7	65.877913	38.99	0.062	65.877913	0.00	0.078	65.877913	0.00	0.312
Instancia 8	39.76383333	0.07	0.156	39.79043333	0.07	0.078	39.79043333	0.07	0.125
Instancia 9	665.12933	0.00	0.063	665.12933	0.00	0.078	665.12933	0.00	0.062
Instancia 10	589.8818205	0.00	0.187	601.6745538	2.00	0.094	589.8818205	0.00	0.125
Instancia 11	138.9281724	0.80	0.562	138.9281724	0.00	0.624	140.0520224	0.81	0.202
Instancia 12	279.757824	0.00	5.132	301.999774	7.95	2.683	279.757824	0.00	1.17
Instancia 13	881.5306385	5.50	5.429	889.0013885	0.85	4.415	932.8077215	5.82	1.216
Instancia 14	2132.239622	0.00	2.481	2232.329789	4.69	1.263	2132.239622	0.00	0.53
Instancia 15	622.7005172	4.71	17.269	676.0701572	8.57	5.476	653.4545717	4.94	0.858
Instancia 16	680.8922983	29.58	1.216	723.4223017	6.25	0.998	681.219365	0.05	0.624
Instancia 17	707.0905877	4.84	16.911	730.3976123	3.30	3.884	743.0591223	5.09	1.217
Instancia 18	557.7429407	2.85	13.275	561.8514407	0.74	8.954	574.0769927	2.93	3.276
Instancia 19	768.12188	0.09	49.39	853.2144853	11.08	12.574	768.841671	0.09	4.291
Instancia 20	284.3712	3.27	9.844	306.9027213	7.92	1.498	293.5649113	3.23	0.858
Instancia 21	343.0836403	2.26	83.71	348.0940323	1.46	52.464	350.997144	2.31	14.414
Instancia 22	404.301613	4.59	3.198	405.722403	0.35	2.059	423.7602005	4.81	0.437
Instancia 23	615.3488435	5.16	5.429	630.1200695	2.40	4.353	648.8221135	5.44	2.309
Instancia 24	480.6140413	0.62	14.711	499.82201	4.00	9.516	483.6359273	0.63	1.997
Instancia 25	1030.669968	0.29	172.366	1131.004064	9.73	67.767	1033.715832	0.30	8.767
Instancia 26	1624.121255	19.09	13.572	1798.340297	10.73	9.017	1664.747338	2.50	2.449
Instancia 27	1923.873541	5.76	148.31	1973.509643	2.58	77.392	2041.337866	6.11	37.846
Instancia 28	1023.920032	6.08	516.067	1088.919042	6.35	236.919	1076.224639	5.11	46.036
Instancia 29	706.9534438	7.23	3312.666	726.5720063	2.78	15.631	763.7521053	8.03	14.524
Instancia 30	1258.600529	1.60	194.049	1320.424309	4.91	113.741	1279.040285	1.62	34.336
Instancia 31	1404.70884	5.68	236.186	1471.982373	4.79	94.599	1478.289679	5.24	37.019
Instancia 32	1078.021976	0.83	46.192	1151.206601	6.79	26.146	1087.028351	0.84	7.254
Instancia 33	2180.80719	11.87	497.69	2415.564737	10.76	143.974	2321.092347	6.43	42.963
Instancia 34	739.635515	9.59	287.307	820.2216	10.90	167.077	818.1024083	10.61	9.235
Promedio Tiempo		**5.04**	**166.28**		**3.95**	**31.28**		**2.44**	**8.07**

Cuadro 4.3: Comparación entre la optimización sin utilizar heuristica, con un gap permitido del 10 % y utilizando heurística con Criterio 2 (gap 10 %). La columna *Gap Relajación* representa el gap obtenido de la primer solución entera de *CPLEX*

Capítulo 5

Conclusiones

En este trabajo se presentaron cuatro modelos originales para el *PCTC*, uno de ellos asociado con el problema de *m-anillo-estrella*, utilizando la misma idea planteada en [1]. En este caso, se puede observar como dos problemas completamente diferentes pueden ser descriptos en un modelo similar. Dos de los modelos planteados aprovechan la estructura de grafos *caterpillar* que presentan las soluciones factibles de este problema. Cabe señalar que en [13] se estudia el problema del *PCTC* donde se desarrollan dos heurísticas que utilizan grafos *caterpillar*. Estos modelos surgieron del análisis de la solución modelando el problema, en algunos casos, con grafos. Se realizaron pruebas empíricas sobre 34 instancias reales y se seleccionó el modelo con mejor performance.

Se propusieron varias familias de desigualdades válidas para el modelo seleccionado. Muchas de estas familias se basan en la estructura y las caraterísticas de los grafos *caterpillar* (como las familias de desigualdades *garra* y *ciclo*). Se propusieron algoritmos de separación sencillos para estas familias, implementando un algoritmo de tipo *branch and cut*. Se realizaron experimentos con distintos valores para los parámetros que determinan la frecuencia de ejecución de los algoritmos de separación y el umbral de valores de las variables consideradas, con la intención de estudiar el balance entre el tiempo dedicado al algoritmo de separación y la eficacia de los cortes hallados. Se concluyó que ninguna familia proporciona beneficios contundentes desde el punto de vista de los tiempos de ejecución.

Una observación importante relacionada con esta última conclusión está dada por el hecho de que –para casi todas las instancias– la función objetivo del óptimo entero coincide con el valor de la relejación lineal (fraccionaria). En este sentido, el agregado de cortes puede ajustar la relajación lineal pero no genera una mejora en el *gap* de optimalidad de la relajación lineal. Por este motivo, se propuso una heurística primal basada en el mismo modelo de programación entera, con el objetivo de generar buenas cotas primales para el problema.

Se propuso una heuristíca de dos fases que parte de una solución fraccionaria. Se construye en primer lugar un grafo parcial utilizando las variables no nulas asociadas con las combinaciones utilizadas y, como este grafo puede no ser un grafo *caterpillar*, se quitan ejes evitando que se formen ciclos y garras. La segunda fase consiste en ejecutar el modelo de programación entera fijando las variables asociadas con ejes de este grafo, con la intención de que al prefijar una parte importante de la solución *CPLEX* pueda encontrar rápidamente una solución factible.

Se observó que si se aplica la heurística con un *gap* del 10 % para el procedimiento global, se obtienen mejoras importantes tanto en cuanto a la calidad de las soluciones halladas como

en cuanto a los tiempos de ejecución. Si bien utilizando este criterio no se llega a la solución óptima, en la mayoría de las instancias el *gap* no supera el 6 %.

Como trabajos futuros, sería interesante estudiar otras ideas para construir heurísticas primales para este problema. Dado que en casi todas las instancias la función objetivo del óptimo de la relajación lineal coincide con la función objetivo del óptimo entero, parece razonable centrar todos los esfuerzos en la construcción de heurísticas tan buenas como sea posible, dado que la relajación lineal proporciona muy buenas cotas duales. En este sentido, es interesante proponer y evaluar nuevas estrategias para fijar variables a partir de la solución fraccionaria, y analizar heurísticas basadas en los otros modelos considerados en este trabajo.

Apéndice A

Implementación

Describimos en este capítulo los detalles de la implementación. Se puede dividir en dos partes: generación y ejecución de los modelos y refuerzos al *branch and bound* especiales para el modelo *caterpillar*.

A.1 Generación y ejecución de los modelos

Los modelos se implementaron en el lenguaje de modelado *zimpl*, que permite generar un archivo .zpl para cada uno. Los siguientes archivos corresponden a la implementación de cada modelo:

- anillo.zpl: Modelo *anillo-estrella*
- caterpillar.zpl: Modelo *caterpillar*
- caterpillarConCortes.zpl: Modelo *caterpillar con cortes*
- modeloCuadratico.zpl: Modelo *cuadrático*

Cada uno de estos modelos en formato .zpl es configurable de acuerdo a una serie de archivos propios que toma como entrada. Se implementó un programa en lenguaje *java* que permite visualizar las instancias y generar dichos archivos. Se utilizó el paquete *CPLEX 12.1* para la resolución de los modelos, y tanto la generación dinámica de cortes como la ejecución de la heurística primal se ejecutan a través de los callbacks de *CPLEX*.

A.1.1 Archivos auxiliares

En esta sección se describen los archivos necesarios para procesar los .zpl para los distintos modelos. En primer lugar, la Figura A.1 muestra el diagrama de clases que representan los parámetros. Dichos objetos son los encargados de generar todos los parámetros que necesita el intérprete zimpl para procesar los archivos .zpl.

Los siguientes archivos se generan como entrada de los .zpl para la generacíon de los modelos correspondientes a los grupos de pedidos de cada instancia:

datos_corrugadora.txt: Guarda el tamaño de la corrugadora. En nuestros experimentos, para todas las pruebas se utilizó un ancho de 1400 mm.

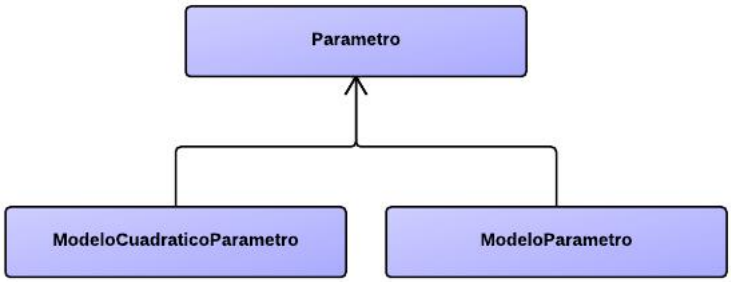

Figura A.1: Diagrama de clases de los parámetros de los modelos

datos_pedidos.txt: Guarda el detalle de cada pedido. El formato es (pedido; atributo; valor), donde **pedido** es el número de pedido, **atributo** puede ser "cantidad", "ancho" o "alto" y **valor** es el valor de dicho atributo.

pedidos.txt: Guarda la cantidad de pedidos dentro de la intancia.

combinacion_data.txt: Guarda todas las combinaciones con el siguiente formato: (pedido1; pedido2; combinacion; cantidad1; cantidad2), donde **pedido1** es el nombre del primer pedido de la combinación, **pedido2** es el nombre del segundo pedido de la combinacion, **combinacion** es la ubicación que tiene la combinacion dentro de la secuencia de combinaciones de la instancia, **cantidad1** es la cantidad de pedido1 y **cantidad2** es la cantidad de pedido2.

max_const.txt: Cota superior en metros lineales que se puede fabricar para una combinación de dos pedidos i y j. Dicho valor se calcula como: máx$(l_i\ c_i/n^i_{ijc}, l_j\ c_j/n^j_{ijc})$.

tamano_combinacion.txt: Este archivo guarda la cantidad de combinaciones para cada par de pedidos i, j.

cantidad_max.txt: Guarda el porcentaje de tolerancia p de la cantidad de pedidos a fabricar.

pertenencia.txt: Cada registro en este archivo tiene el siguiente formato: (nombre de la combinación; pedido1; pedido2). La primer posición es la combinación, la segunda y la tercer posición se corresponden al primer y el segundo pedido que pertencen a dicha combinación. Este archivo sirve para indicar en qué combinación está cada pedido.

A.1.2 Algoritmos asociados

El Algoritmo 2 genera todas las combinaciones posibles para un grupo de pedidos con un ancho de corrugadora y un refile permitido. Utiliza como sub-rutina al Algoritmo 3, que construye todas las combinaciones entre dos pedidos dados. Tanto para el modelo *anillo-estrella*, *caterpillar* y *caterpillar con cortes* se utiliza la clase *AnilloParametros* y para el modelo *cuadrático* la clase se llama *Modelo3Parametros*.

```
Algorithm: generarTodasLasCombinaciones
Input: pedidos
Output: combinaciones
begin
    for p1 ∈ pedidos do
        for p2 ∈ pedidos do
            combinaciones.agregar(generar(p1, p2, ancho, refile)
        end
    end
end
```

Algorithm 2: generarTodasLasCombinaciones(pedidos)

```
Algorithm: generar
Input: pedido1, pedido2, anchoCorrugadora, refile
Output: combinaciones
begin
    anchoCorrugadoraConRefile ← (1 − refile) ∗ anchoCorrugadora
    if pedido1 == pedido2 then
    c ← (pedido1, pedido2, anchoCorrugadora/anchoPedido1, 0)
    else
        cantidadPedido1 ← 0
        cantidadPedido2 = 0
        while anchoPedido1 ∗ cantidadPedido1 + anchoPedido2 ∗ cantidadPedido2 ≥
        anchoCorrugadora do
            while
            anchoPedido1 ∗ cantidadPedido1 + anchoPedido2 ∗ cantidadPedido2 ≥
            anchoCorrugadora do
                if
                anchoPedido1 × cantidadPedido1 + anchoPedido2 ∗ cantidadPedido2 ≥
                anchoCorrugadora ∧ anchoPedido1 × canitdadPedido1 +
                anchoPedido2 × cantidadPedido2 ≥ anchoCorrugadoraConRefile
                then
                    c ← (pedido1, pedido2, cantidadPedido1, cantidadPedido2)
                end
                cantidadPedido2 + +
            end
            cantidadPedido2 = 0
            cantidadPedido1 + +
        end
    end
end
```

Algorithm 3: generar(pedido1, pedido2, anchoCorrugadora, refile)

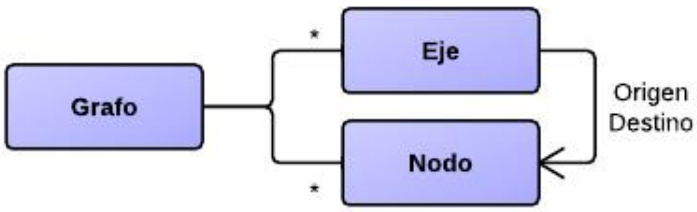

Figura A.2: Diagrama de clases para representar grafos.

A.2 Algoritmos Generales

A.2.1 Armado de un grafo a partir de una solución

Este algoritmo se utiliza en todos los callbacks que requieren de un grafo para deducir un corte de alguna familia de desigualdades válidas. El grafo se arma a partir de las variables e^x_{ij} y e^y_{ij} o la variable m_{ijc} (sin tener en cuenta el subíndice c). Es posible pasar como parámetro al algoritmo una cota inferior al valor que deben cumplir dichas variables, de esta forma el algoritmo es utilizado tanto para los procedimientos de separación (agregado de *user contraints*) como para la generación dinámica de restricciones a partir de soluciones enteras pero no factibles (agregado de *lazy constraints*). Dicho algoritmo se encuentra en la clase *ProblemResolver* y el detalle de la implementación se muestra en el Algoritmo 4.

Algorithm: armarGrafo
Input: cplex, xs, xMinValue
Output: Grafo
begin
  for $v \in lp.getNumVars()$ **do**
    if $(v == e^x_{ij} \vee v == e^y_{ij}) \wedge (cplex.getValue(v) \geq xMinValue)$ **then**
      $grafo \leftarrow agregarEje(v)$
    end
  end
end

Algorithm 4: armarGrafo(cplex, xs, xMinValue)

A.2.2 Otros algoritmos

La Figura A.2 contiene el diagrama de clases del modelado para representar *grafos* dentro de la implementación. Cada eje del grafo además de guardar el nodo origen i y el nodo destino j guarda la posición p que tiene la variable en el array **xs**, esto es para que al iterar el grafo encontrar la variable asociada sea en O(1).

Se implementaron varios algoritmos útiles para operar con grafos, que se encuentran en la clase *Grafo*. Estos algoritmos son utilizados a lo largo de toda la implementacíon para los procedimientos de separación, la heurística primal y el agregado de *lazy constraints*.

El Algoritmo 5 consiste en buscar los vecinos de un nodo y los vecinos de esos vecinos para determinar si el grafo contiene una garra. Dicho algoritmo es utilizado para buscar los nodos garras para la familia de desigualdades *garra* (Algoritmo 17) y para la heurística primal (Algoritmo 21). En este contexto, un grafo tiene una garra si existe un nodo con tres vecinos y cada vecino tiene al menos un vecino.

Otro algoritmo muy utilizado en la implementación es el Algoritmo 6 que retorna todas las componentes conexas de un grafo. En general, se utiliza este algoritmo para estudiar cada componente conexa por separado y buscar ciclos o garras según corresponda. Dicho algoritmo utiliza un procedimiento auxiliar (ver Algoritmo 7) que busca para cada nodo en particular la componente conexa a la que pertenece. Para determinar si una componente conexa tiene un ciclo se utiliza el Algoritmo 8, utilizado en el algoritmo de separación para la familia de desigualdades *ciclo* (ver Algoritmo 16) y para determinar si una solución tiene asociado un grafo *caterpillar* (ver Algoritmo 15). En el caso del modelo *caterpillar con cortes* además de verificar que no existen ciclos tambien se debe verificar que no haya garras. La forma de hacer esto es para cada componente conexa quitar los nodos hoja utilizando el Algoritmo 10 y luego verificar que el resultado es un grafo con un camino utilizando el Algoritmo 9.

Algorithm: buscarNodosGarra
Input: nodo
Output: garra
begin
  $adyacentes \leftarrow getAdyacentes(nodo)$
  for $n \in adyacentes$ **do**
    if $n.grado \geq 2$ **then**
      $garra \leftarrow n$
      $garra \leftarrow$ vecino de $n \neq nodo$
    end
    if($garra.size \geq 6$) return garra
  end
end

Algorithm 5: buscarNodosGarra(nodo)

Algorithm: getComponenteConexa
Input:
Output: componentesConexa: componentes conexas del grafo
begin
  for $n \in nodos$ **do**
    if *n no esta en alguna componente conexa* **then**
      $componentesConexa \leftarrow dameComponenteConexa(n)$
    end
  end
end

Algorithm 6: getComponenteConexa(nodo)

Algorithm: dameComponenteConexa
Input: nodo n
Output: componenteConexa: componenente conexa al que pertenece n
begin
  for $ady \in adyacentes(n)$ **do**
    $componenteConexa \leftarrow agregar(ady)$
    $componenteConexa \leftarrow agregar(dameComponenteConexa(ady))$
    marcar ady como que pertenece a una componenteConexa
  end
end

Algorithm 7: dameComponenteConexa(nodo): Algoritmo auxiliar para buscar las componentes conexas (*getComponenteConexa*)

Algorithm: tieneUnCiclo
Input:
Output: **true** si tiene un ciclo
begin
  $getEjes().size() > getNodos().size() - 1$
end

Algorithm 8: tieneUnCiclo

Algorithm: esCamino
Input:
Output: **true** si el grafo es un camino
begin
  $cantidadNodosGrado1 \leftarrow 0$
  $cantidadNodosGrafoMayor2 \leftarrow 0$
  for *n nodo del grafo* **do**
    if $grado(n) == 1$ **then**
      $cantidadNodosGrado1++$
    end

    if $grado(n) > 2$ **then**
      $cantidadNodosGradoMayor2++$
    end

  end
  return $cantidadNodosGrado1 == 2 \wedge cantidadNodosGradoMayor2 == 0) \vee nodos.size == 1$
end

Algorithm 9: esCamino

```
Algorithm: getGrafoSinHojas
Input:
Output: Grafo sin hojas
begin
    for n nodo del grafo do
        if grafo(n) == 1 then
            quitarNodo(n)
        end
    end
end
```

Algorithm 10: getGrafoSinHojas

A.3 Algoritmo branch and cut

Existen varios puntos de control abiertos en *CPLEX* de forma tal que el usuario puede agregar lógica para configurar la resolución para un problema particular. La jerarquía de callbaks se puede ver en la Figura A.3.

En este trabajo se utilizaron cuatro callbacks de diferentes tipos:

- *LazyConstraintCallback*: Usado para agregar nuevas restricciones si el grafo resultado no es un caterpillar, se aplica cuando *CPLEX* encuentra una solución factible. Se usa para el modelo *Caterpillar* y *Caterpillar con Cortes*.
- *IncumbentCallback*: Usado para rechazar soluciones que no generan grafos caterpillar.
- *UserCutCallback*: Usado para aplicar cortes en soluciones fraccionarias.
- *HeuristicCallback*: Usado para ayudar a *CPLEX* a encontrar un grafo caterpillar.

A.3.1 LazyConstraintCallback: Ciclo y Garra

Este callback se usa en el modelo *Caterpillar* para evitar que se formen ciclos y en el modelo *Caterpillar con Cortes* para evitar garras y ciclos. La implementacion de dicho algoritmo se puede ver en 11. Notar, que se utiliza dos algorimos auxiliares 12 y 13, el primer algoritmo agrega las restricciones de *ciclo* y es aplicado tanto para el modelo *Caterpillar* como para el modelo *Caterpillar con cortes*, mientras que el segundo algoritmo agrega las restricciones que impiden que el grafo solución tenga garras y sólo aplica para el modelo *Caterpillar con cortes*. El diagrama de clases asociado se puede se puede ver en la Figura A.4.

```
Algorithm: main
begin
    buscarCortesCiclo()
    addSpecificLazyConstraint()
end
```

Algorithm 11: CaterpillarLazyConstraintCallback

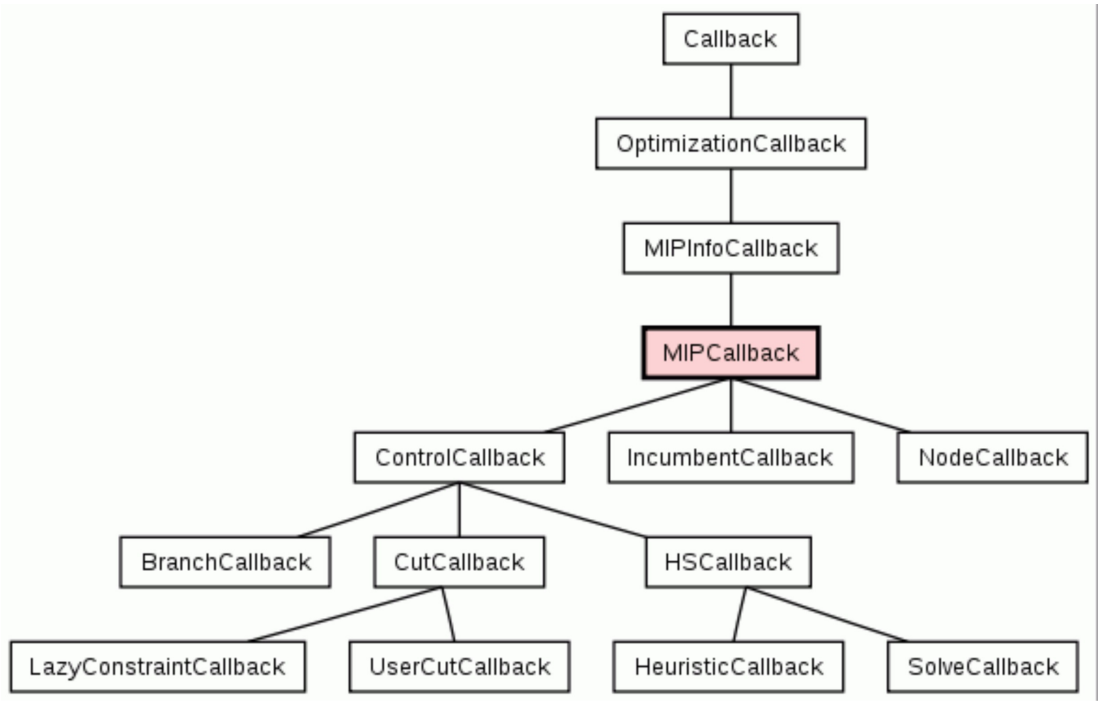

Figura A.3: Jerarquía de callbacks de *CPLEX*

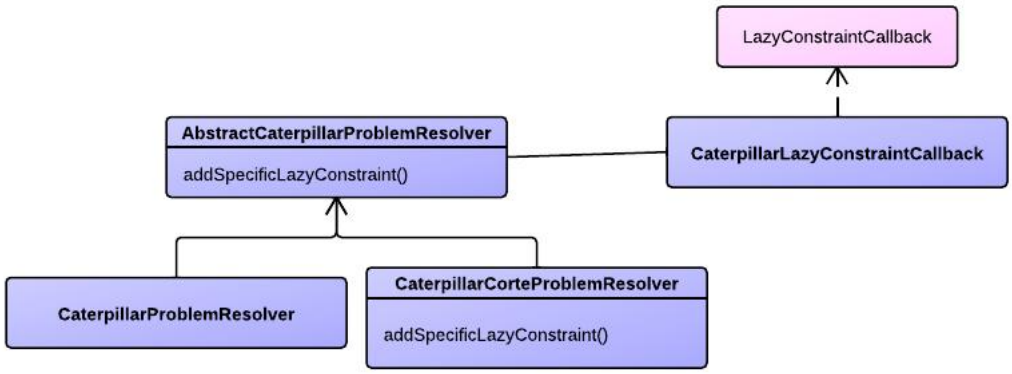

Figura A.4: Jerarquía de clases para las LazyConstraintCallback utilizadas

Algorithm: buscarCortesCiclo
begin
  $xs \leftarrow solucionFactibleCplex$
  $grafo \leftarrow ArmarGrafo(cplex, xs, 1)$
  if *grafo.tieneCiclo()* **then**
    crear restriccion usando los ejes del grafo.
  end
end

Algorithm 12: buscarCortesCiclo

Algorithm: addSpecificLazyConstraint
begin
  $garra \leftarrow grafo.buscarNodoGarra()$
  armar nueva restriccion teniendo en cuenta los ejes en *garra*
end

Algorithm 13: addSpecificLazyConstraint

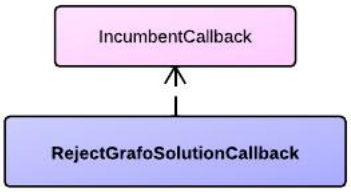

Figura A.5: Jerarquía de clases para las IncumbentCallback utilizadas

A.3.2 IncumbentCallback: Validación del grafo

Despues de que *CPLEX* encuentra una solución entera se aplica un algoritmo que determina si el grafo resultado cumple con ser un grafo *caterpillar*. Este tipo de callback se usa para rechazar soluciones que no cumplen con tener un grafo *caterpillar* asociado. La clase que implementa este callback se llama *RejectGrafoSolutionCallback* y el pseudocódigo se detalla en el Algoritmo 14. Notar que, al igual que en el caso anterior, se separó la implementación para el modelo *caterpillar* y el modelo *caterpillar con cortes*: para el primero se valida que cada componente conexa no tenga ciclos, y para el segundo se válida que adémas de no tener ciclos no contenga garras (ver Algoritmo 15). El diagrama de clases asociado se puede ver en la Figura A.5

Algorithm: main
begin
  $grafo \leftarrow armarGrafo(cplex, xs, 1)$
  for $s \in grafo.getComponentesConexas()$ **do**
    if $s.tieneUnCiclo() \wedge hayQueRejectar()$ **then**
      rechazar la solución.
    end
  end
end

Algorithm 14: RejectGrafoSolutionCallback

A.3.3 UserCutCallback: Cortes

La Figura A.6 muestra la jerarquía de clases que se utilizó para representar los distintos callabacks de este tipo.

Algorithm: hayQueRejectar
Input: grafo
Output: **true** si hay que rechazar el grafo
begin
  $camino \leftarrow grafo.getGrafoSinHojas()$
  camino.esCamino()
end

Algorithm 15: hayQueRejectar

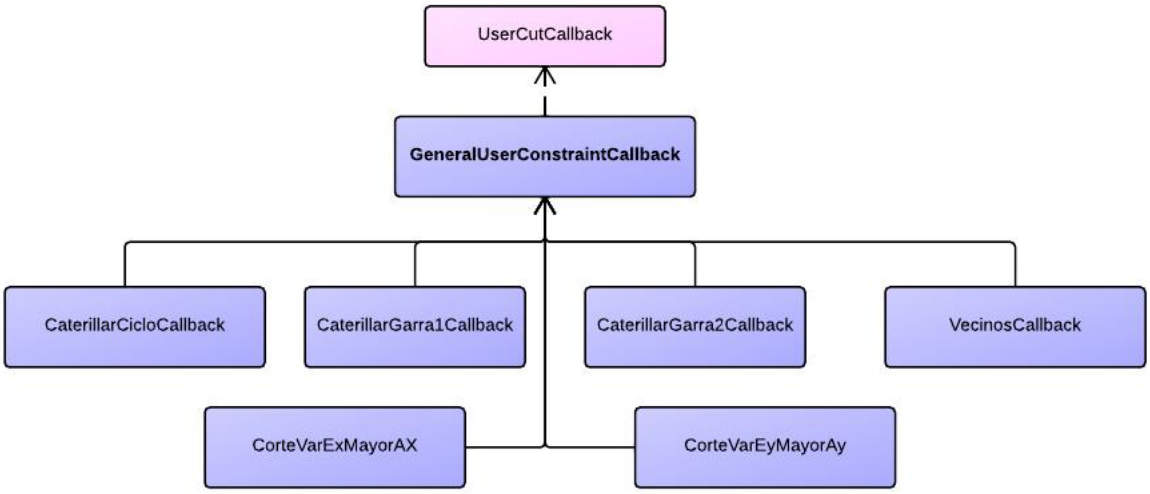

Figura A.6: Jerarquía de clases para las UserCutCallback utilizadas

Corte ciclo

El algoritmo de búsqueda y armado de las restricciones *ciclo* es similar al utilizado para las lazyConstraintCallback (ver Algoritmo 11). En este caso se arma el grafo teniendo en cuenta las variables e_{ij}^x y $e_i^y j$ mayores a 0 (ver el Algoritmo 16, implementado en la clase *CaterillarCicloCallback*).

```
Algorithm: main
begin
    grafo ← armarGrafo(cplex, xs, 0) if grafo.hayUnCiclo then
        agregar constraint con los ejes de grafo
    end
end
```

Algorithm 16: CaterpillarCicloCallback

Corte garra

Se utiliza la misma idea para los dos algoritmos de separación de los cortes garra. Se buscan nodos garras y se construyen las restricciones dependiendo del caso. Se utilizan las clases *CaterpillarGarra1Callback* y *CaterpillarGarra2Callback* correpondientes a las familia de desigualdades (3.1) y (3.2), respectivamente. El Algoritmo 17 detalla la implementación.

```
Algorithm: main
begin
    grafo ← armarGrafo(cplex, xs, 0)
    for n nodo del grafo do
        garra ← grafo.buscarNodosGarra(n)
        if garra.size() ≥ 6 then
            armar restriccion con los ejes y agregarla a cplex
        end
    end
end
```

Algorithm 17: CaterpillarGarra1Callback

Corte vecino

La clase *VecinosCallback* implementa este callback. Se puede ver el detalle de la implementaciíon en el Algoritmo 18.

Corte nodos internos y externos

Las clases *CorteVarExMayorAX* y *VarEyMayorAy* implementan el callback para laS familia de desigualdades (3.5) y (3.6), respectivamente. Notar que en este caso no se arma el grafo asociado a la solución intermedia.

Algorithm: main
begin
  $grafo \leftarrow armarGrafo(cplex, xs, 0)$
  buscar las variables ex y ey que se corresponden a los ejes de $grafo$
  buscar las variables x_i para cada nodo i del $grafo$
  armar restriccion y agregarla
end

Algorithm 18: CaterillarNoTodosLosVecinosYCallback

Algorithm: main
begin
  buscar todas las variables ex mayores a 0
  buscar las variables xs
  armar restriccion y agregarla si es violada
end

Algorithm 19: CorteVarExMayorAX

Algorithm: main
begin
  buscar todas las variables ey mayores a 0
  buscar las variables ys
  armar restriccion y agregarla si es violada
end

Algorithm 20: VarEyMayorAy

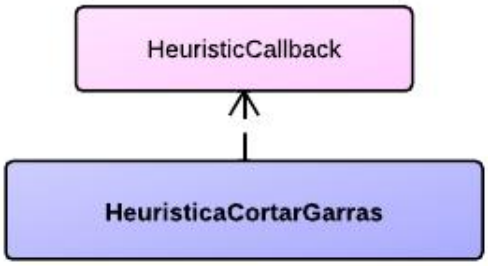

Figura A.7: Jerarquía de clases para las HeuristicCallback

A.3.4 HeuristicCallback

Para la heurística se arma el grafo con $k > 0$, se quitan las garras y los ciclos. Luego, se arma un nuevo problema y se agregan nuevas restricciones seteando las variables e_{ij}^x y e_{ij}^y que tengan valores iguales a 1. Se optimiza el problema secundario y si la solución es óptima se setean las variables resultantes de la optimización al problema original. La jerarquía de clases utilizada se detalla en la Figura A.7, la clase que implementa dicho callback se llama *HeuristicaCortarGarras2* y el algoritmo utilizado se detalla en el Algoritmo 21. El algorimo utiliza las rutinas auxiliares descriptas en el Algoritmo 23 y el Algoritmo 22, encargadas de quitar ejes de ciclos o garras para que el grafo resultado sea *caterpillar*.

Algorithm: main
begin
  $grafo \leftarrow armarGrafo(cplex, xs, 0)$
  for $s \in grafo.getComponentesConexas()$ **do**
    quitarGarras(s)
    quitarCiclos(s)
  end
  $nuevoCplex \leftarrow newcplex()$
  for e *eje de* $grafo$ **do**
    $nuevoCplex \leftarrow$ agregar restriccion eje $ex_{ij} == 1$ y $ey_{ij} == 1$
  end
  if *nuevoCplex es optimo* **then**
    setear variables al cplex original
  end
end

Algorithm 21: HeuristicaCortarGarras

```
Algorithm: quitarGarras
Input: grafo
Output: grafo sin garras
begin
    for n nodo de grafo do
        ejeGarra ← grafo.buscarNodoGarra(n)
        grafo.quitarEje(ejeGarra)
    end
end
```

Algorithm 22: quitarGarras

```
Algorithm: quitarCiclos
Input: grafo
Output: grafo sin ciclos
begin
    while grafo.tieneUnCiclo() do
        ejesConCiclo ← grafo.buscarCiclos()
        grafo.quitarEje(ejesConCiclo)
    end
end
```

Algorithm 23: quitarCiclos

Apéndice B

Programa PCTC

Aprovechando que la solución al PCTC se puede visualizar fácilmente por medio de un (pseudo)grafo que representa las combinaciones y buscando una forma sencilla de testear las instancias, se diseñó un programa para mantener las instancias de prueba, interactuar con cplex y analizar las soluciones. La Figura B.1 muestra la interfaz del programa con sus componentes enumeradas de 1 a 8:

1. **Lista de instancias**: Lista con todas las instancias que fueron estudiadas en este trabajo.
2. **Refile**: Posibles refiles. En este caso son 0.05, 0.1 y 1.
3. **Optimizaciones**: Lista con los agregados (agregado de cortes y aplicación de la heurística primal) que se pueden aplicar al modelo *caterpillar*.
4. **Modelo**: Modelos estudiados (*anillo-estrella*, *caterpillar*, *caterpillar con cortes* y *cuadrático*).
5. **Consola**: Muestra el log de cplex cuando se selecciona la opción de resolver el modelo. Al terminar el proceso se muestra el valor final de todas las variables.
6. **Visor de desigualdades**: Al seleccionar una familia de desigualdades, se muestra aquí su expresión.
7. **Visor del detalle de la instancia**: Al seleccionar una instancia se muestra el detalle de todos los pedidos con su alto, ancho y la cantidad solicitada.
8. **Visor de grafo**: Muestra el grafo que representa la solución hallada y la configuración seleccionada. Esta opción no está disponible para el modelo *cuadrático*.

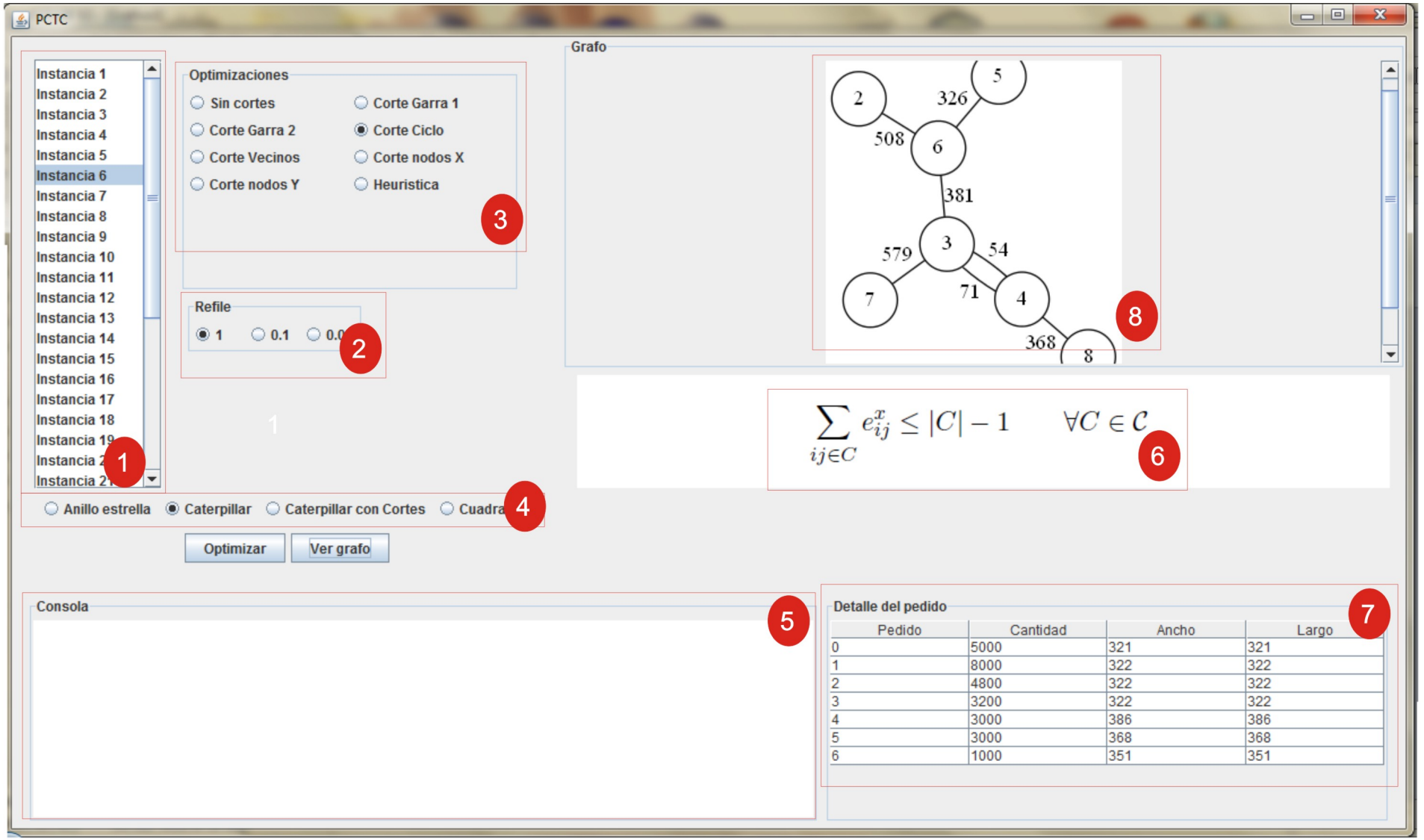

Figura B.1: Programa PCTC

Para ejecutar el programa es necesario tener *Cplex* y *Java* instalado. Para esto, en el directorio donde se encuentra el archivo .jar escribir por línea de comandos:

java -jar ptct

Bibliografía

[1] H. Berinsky, *Un modelo y algoritmo branch-and-cut para el problema del m-anillo-estrella con capacidades*. Tesis de Licenciatura en Ciencias de la Computación, Departamento de Computación, FCEyN, UBA (2010).

[2] A. Bolat, *An extended scheduling model for producing corrugated boxes*. International Journal of Production Research 38 (2000) 1579–1599.

[3] Ecsabox, *Principios básicos del empaque corrugado*. http://www.ecsabox.com/principios_del_carton.pdf, 2012.

[4] P.C. Gilmore, R. E. Gomory, *A Linear Programming Approach to the Cutting Stock Problem*, Oper. Res., 9 (1961), 849-859.

[5] P.C. Gilmore, R. E. Gomory, *A linear programming approach to the cutting-stock problem - Part II*. Operations Research 11 (1963) 863-888.

[6] H. Hideki, M. Pinto, *An integrated cutting stock and sequencing problem*. European Journal of Operational Research 183 (2007) 1353–1370.

[7] IBM. *IBM ILOG CPLEX Optimization Studio V12.3*.

[8] L.V. Kantorovich, *Mathematical Methods of Organizing and Planning Production*, Management Sci., 6 (1960), 366-422.

[9] C. Lekkerkerker, J. Boland. *Representation of a finite graph by a set of intervals on the real line*, Fundam. Math. 51 (1962), 45–64.

[10] A. Linhares and H. Hideki, *Connections between cutting-pattern sequencing, VLSI design, and flexible machines*. Computers and Operations Research 29 (2002) 1759–1772.

[11] K. Matsumoto, H. Miwa, and T. Ibaraki, *Scheduling of corrugated paper production*. European Journal of Operational Research 192 (2009) 782–792.

[12] Phpsimplex, *History of Operations Research*. http://www.phpsimplex.com/en/history.htm, 2012.

[13] F. Rinaldi, A. Franz. *A two-dimensional strip cutting problem with sequencing constraint*. European Journal of Operational Research 183 (2007) 1371–1384.

[14] G. Wäscher, H. Haußner, H. Schumann. *An Improved Typology of Cutting and Packing Problems*. European Journal of Operational Research Volume 183, Issue 3, 1109-1130

[15] Wikipedia, *Programación Lineal*. http://es.wikipedia.org/wiki/Programacion_lineal, 2012.

[16] Wikipedia, *Bloqueo de Berlín*. http://es.wikipedia.org/wiki/Bloqueo_de_Berlin, 2012.

[17] L. Wolsey. *Integer Programming*, John Wiley & Sons, 1998.