

Problema de los Visitadores

Tesis de Licenciatura

Santiago Santucho
Juan Pablo Sturla

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

Noviembre de 2010

A nuestras familias y amigos

Agradecimientos

A Paula por sus conocimientos y su dedicación durante el desarrollo de la Tesis y a nuestras familias que nos aguantaron durante todos estos años de carrera.

Resumen

Tenemos un conjunto de *comercios* que deben ser visitados y para esto contamos con un grupo de personas que serán las encargadas de visitarlos. A estas personas las llamaremos *visitadores*. Debemos asignarle a cada visitador un conjunto de comercios, con un orden determinado, para que visite con el fin de realizar venta de combos, colocación de materiales promocionales, relevamiento de precios, etc. Tanto la cantidad de comercios como la cantidad de visitadores son datos del problema y cada comercio debe ser visitado por un único visitador.

Como los visitadores reciben comisiones por las ventas, es necesario que la cantidad de comercios asignados a cada uno de ellos sea equitativa. Además se debe tener en cuenta que la distancia total recorrida por los visitadores sea pequeña ya que el trabajo se realiza caminando (y en muy pocas ocasiones se realizan viajes en colectivo). Por último es necesario que la distancia que recorre cada visitador no sea muy diferente a la del resto para que ninguno de ellos se vea perjudicado por el tiempo necesario para cumplir con su recorrido.

Este problema, al cual llamaremos *Problema de los Visitadores (PV)*, pertenece al conjunto de problemas de Optimización Combinatoria que surge del estudio de un caso real y que puede ser trasladado a muchas otras aplicaciones.

PV es una variación del problema del múltiple viajante de comercio. En el problema de los visitadores agregamos la restricción de balanceo entre la cantidad de comercios visitados por cada visitador y, en un principio, planteamos el objetivo de que la unión de los caminos recorridos por todos los visitadores sea lo más corto posible. Luego cuando comenzamos a formalizar el modelo (que se detallará más adelante) notamos que la distancia recorrida por un visitador podía diferir mucho de la distancia recorrida por otro. Para evitar este problema planteamos un segundo objetivo que consiste en minimizar la sumatoria de las diferencias de distancias entre los recorridos de todos los visitadores entre sí.

Como muchos de los problemas de Optimización Combinatoria, el problema de los visitadores puede ser modelado mediante formulaciones de programación lineal entera o entera mixta. Los algoritmos Branch-and-Cut son parte de las herramientas más efectivas que se conocen para resolver un modelo de programación lineal entera.

El objetivo de la tesis es estudiar el problema de los visitantes desde un enfoque de programación lineal entera, proponer una formulación para modelarlo y desarrollar e implementar un algoritmo Branch-and-Cut para resolver el problema planteado.

Índice general

1. Introducción	1
1.1. Definición del Problema de los Visitadores	1
1.2. Objetivo de la Tesis	2
2. Programación Lineal Entera	5
2.1. Definición	5
2.2. Formulaciones	5
2.3. Algoritmos para Problemas de Programación Lineal Entera Mixta	7
2.3.1. Algoritmos <i>Branch-and-Bound</i>	7
2.3.2. Algoritmos de Planos de Corte	9
2.3.3. Algoritmos Branch-and-Cut	12
3. Modelo de Programación Lineal Entera para el PV	15
3.1. Modelo	15
3.2. Análisis de la función objetivo	19
4. Implementación del algoritmo Branch-and-Cut	21
4.1. Detección de ciclos	21
4.1.1. Variantes FB2 y FB3	23
4.1.2. Variante de optimización de subtour	24
4.2. Heurística Inicial (HI)	26
4.2.1. Pseudocódigo de la Heurística Inicial	28
4.3. Variables X y V	29
4.3.1. Comparación de desempeño	29
4.4. Heurística Primal	31
4.4.1. Heurística Primal en Serie	32
4.4.2. Heurística Primal en Paralelo 1	33
4.4.3. Heurística Primal en Paralelo 2	35
4.5. Rompimiento de Simetría	39
4.5.1. Comparación de desempeño	41
4.6. Desigualdades válidas	42
4.6.1. Desigualdad de Capacidad	42

4.6.2. Desigualdades <i>Star</i>	42
4.6.3. Desigualdades <i>Comb</i>	43
4.6.4. Algoritmos de Separación	43
4.6.5. Evaluación de las Familias de Desigualdades Válidas	44
4.7. Comparación CPLEX vs BC-V	47
4.8. Conclusiones y trabajo futuro	48
Bibliografía	49

Capítulo 1

Introducción

1.1. Definición del Problema de los Visitadores

A continuación detallaremos algunos puntos importantes a tener en cuenta para comprender el problema que vamos a resolver:

- Tenemos una cantidad fija C de comercios a visitar (la cantidad de comercios es fija a la hora de armar los recorridos, pero con el paso del tiempo se van produciendo altas y bajas de comercios, con lo cual se deben recalculer los recorridos periódicamente). Estos comercios pertenecen al menos a alguna de las siguientes categorías: farmacia, perfumería, autoservicio, pañalera, almacén tradicional, minimercado y veterinaria.¹
- Tenemos una cantidad fija V de visitadores (esta cantidad puede variar en función de las variaciones en la cantidad de comercios pero la cantidad siempre es fija a la hora de armar los recorridos).
- La distancia entre los comercios es la distancia euclídeana para evitar tener que calcular dichas distancias basándonos en algún sistema cartográfico. Cabe aclarar que esta decisión no simplifica el problema a solucionar ya que solamente habría que modificar el cálculo inicial de las distancias.
- A la hora de armar el recorrido de los visitadores no se tiene en cuenta el domicilio de cada uno. Por ejemplo, podría darse el caso que un visitador tenga que recorrer 1 cuadra para llegar al primer comercio de su recorrido y que otro tenga que recorrer 20 cuadras (lo mismo vale para el regreso desde el último comercio a la casa) ya que dichas distancias no serán tenidas en cuenta a la hora de calcular la distancia total de su recorrido.

¹Dada que el Problema de los Visitadores es un problema real, hemos mencionado los tipos de comercio a visitar.

- Algunas de las tareas a realizar por los visitantes en cada comercio son:
 - Venta de combos (conjuntos de productos predefinidos a un precio promocional)
 - Colocación de materiales promociones (POP).
 - Relevamiento de precios.
 - Relevamiento de encuestas.
 - Acomodamiento de góndolas, vidrieras y exhibidores.

En una segunda etapa, estamos interesados en generalizar el problema, permitiendo considerar que los visitantes y los clientes no son homogéneos. Es decir, no todos los visitantes trabajan la misma cantidad de horas y no todos tienen asignada la misma cantidad de clientes. También podríamos tener clientes que sólo pueden ser atendidos por algunos visitantes determinados para cada uno de ellos.

1.2. Objetivo de la Tesis

Los problemas de Optimización Combinatoria aparecen en la vida real en una gran variedad de disciplinas, desde genética, física, química hasta finanzas, marketing y la industria. Generalmente, estos problemas son fáciles de formular matemáticamente, pero computacionalmente difíciles de resolver.

Para resolver un problema necesitamos de un algoritmo, es decir de un procedimiento sin ambigüedades que consiste de una sucesión finita de pasos a realizar en un orden específico. Para los problemas de Optimización Combinatoria, la enumeración de los elementos de F (conjunto de soluciones factibles del problema) constituye un algoritmo. Sin embargo, si el cardinal de F es grande, es claro que éste no es un método práctico ni eficiente. Es fácil programar un algoritmo de enumeración completa para el Problema de los Visitadores. Sin embargo, una instancia con 100 comercios y 10 visitantes tiene tantas soluciones que no viviríamos lo suficiente para ver la respuesta. Esto casi parece decir que la enumeración completa no resuelve el problema.

¿Cuándo consideramos a un algoritmo eficiente? La teoría de complejidad fue iniciada por Cook en 1971 y estableció criterios para decidir si un algoritmo resuelve un problema de manera eficiente. Se considera que un algoritmo es eficiente si encuentra la solución de tal manera que, en el peor de los casos, el tiempo requerido (número de operaciones elementales) está acotado por un polinomio en la medida de los datos de entrada. Son los llamados algoritmos polinomiales. Hay problemas para los cuales se dispone de un algoritmo polinomial. Estos conforman la clase P. Tal es el caso del problema de flujo máximo, el camino más corto entre dos ciudades, entre otros.

Muchos problemas de Optimización Combinatoria pertenecen a una clase de problemas aparentemente difíciles desde el punto de vista computacional. Esta clase es denominada NP-Difícil en la teoría de la complejidad. Para estos problemas no se conoce aún un algoritmo que encuentre la solución en tiempo polinomial. Sin embargo, si para alguno de estos problemas se encontrara un algoritmo polinomial, esto implicaría que muchos otros problemas también podrían ser resueltos en tiempo polinomial. Entre los problemas de la clase NP-Difícil, podemos mencionar el Problema de Coloreo de Grafos (encontrar la cantidad mínima de colores que se asignan a los vértices de un grafo de tal manera que a dos vértices conectados por una arista no le corresponde el mismo color), el Problema de Máximo Conjunto Independiente (encontrar la mayor cantidad de vértices de un grafo donde todo par de vértices no está conectado por una arista) y el Problema de Camino Hamiltoniano (encontrar la manera de recorrer todos los vértices de un grafo pasando una y sólo una vez por cada uno).

El Problema de los Visitadores pertenece a la clase de problemas NP-Difícil (el Problema del Viajante de Comercio se reduce al PV con un sólo visitador). Muchos de los problemas de Optimización Combinatoria pueden ser modelados mediante formulaciones de programación lineal entera o entera mixta, como por ejemplo la asignación de la tripulación a la flota de aviones de una aerolínea, el diseño de líneas de producción, la asignación de recursos, la asignación de frecuencias radiales y control de inventario, donde personas, máquinas, actividades, recursos, aviones, etc., son indivisibles. En estos modelos el objetivo es buscar el óptimo de una función lineal donde algunas o todas las variables están restringidas a ser enteras y deben verificar un sistema de desigualdades lineales.

La versatilidad dada por los modelos de programación lineal entera hace que esta área tenga gran importancia dentro de la Optimización Combinatoria. Si bien el problema general de programación entera pertenece a la clase NP-Difícil, se ha invertido mucho esfuerzo en el desarrollo de algoritmos competitivos. Durante los últimos 30 años, se produjo un gran progreso en las técnicas capaces de resolver exitosamente estos problemas. Especialmente las basadas en combinatoria poliedral han permitido incrementar el tamaño de las instancias resueltas. El área combinatoria poliedral estudia cómo describir la cápsula convexa del conjunto de soluciones factibles de un problema de programación lineal entero. El trabajo inicial en este sentido fue hecho por Dantzig, Fulkerson y Johnson en 1954, quienes desarrollaron un método para resolver el Problema del Viajante de Comercio.

A comienzos de los '80 se comenzó a aplicar una metodología mixta que conjuga los algoritmos y las ideas que se habían desarrollado hasta entonces dando origen a los llamados algoritmos Branch-and-Cut. De esta manera se lograron resolver exitosamente instancias de tamaño considerable de una gran cantidad de problemas de programación lineal entera, como por ejemplo el Problema de Viajante de Comercio, el Problema de Ordenamiento lineal, el Problema de Corte Máximo, etc.

El objetivo de esta tesis es el desarrollo de un algoritmo tipo Branch-and-Cut para la

resolución del Problema de los Visitadores. Con este fin, introduciremos una nueva formulación de programación lineal entera. Desarrollaremos un algoritmo de planos de corte para resolver el PV. Finalmente, el procedimiento de planos de corte será incorporado a un algoritmo Branch-and-Cut.

El trabajo está organizado de la siguiente manera. En el capítulo 2 introducimos conceptos básicos de programación lineal entera y describimos los principales algoritmos utilizados para su resolución. En el capítulo 3 presentamos nuestro modelo. En el capítulo 4 detallamos los procedimientos y diferentes alternativas que fueron consideradas para desarrollar el algoritmo Branch-and-Cut para resolver el PV. El comportamiento del algoritmo es analizado sobre instancias generadas al azar y sobre instancias creadas ad-hoc. Finalizamos el capítulo comparando nuestro algoritmo con un algoritmo de propósito general, en nuestro caso Cplex. La tesis termina describiendo las conclusiones del trabajo en el capítulo 5.

Capítulo 2

Programación Lineal Entera

2.1. Definición

En las últimas décadas, el uso de modelos de programación lineal entera mixta para resolver problemas de Optimización Combinatoria se ha incrementado enormemente. Avances en el estudio de modelos, en el software, hardware y nuevos algoritmos, permitieron un importante avance en la resolución de problemas que pueden formularse como problemas de programación entera, abarcando problemas de diversas áreas como ser problemas de telecomunicaciones, economía, producción, logística, etc., tales como problemas de ruteo de vehículos, de ruteo y diseño de redes de telecomunicaciones, de administración de cadenas de suministro, de asignación de tripulación en líneas aéreas, asignación de horarios, asignación de tareas en máquinas, minimización de desperdicio en el corte materiales, etc. Día a día continúan surgiendo nuevas aplicaciones derivadas de la necesidad de tomar decisiones en muchos ámbitos que requieren de la formulación de nuevos modelos y desarrollo de algoritmos para resolverlos.

En este capítulo se presentan algunos conceptos básicos de programación entera. En la primera sección sobre formulaciones se describe la formalización de un problema como problema de programación entera. En las secciones de algoritmos Branch-and-Bound, algoritmos de planos de corte y algoritmos Branch-and-Cut se describen métodos utilizados que permiten resolver estos modelos, es decir encontrar sus soluciones óptimas.

2.2. Formulaciones

Mediante un problema de programación lineal entera mixta se pueden modelar situaciones donde se debe minimizar una función lineal sujeta a un conjunto de restricciones, también lineales, donde algunas, o todas, las variables sólo pueden tomar valores enteros. Este es el caso del Problema del Viajante de Comercio, problemas en redes, problemas de

asignación de recursos, problemas de teoría de grafos, y muchísimos otros problemas de optimización combinatoria provenientes de una gran cantidad de disciplinas.

Un problema de programación lineal entera mixta (PEM) puede ser formulado de la siguiente manera:

$$\begin{aligned}
 & \text{Minimizar} && \sum_{j \in I} c_j x_j + \sum_{j \in C} c_j x_j \\
 & \text{sujeto a} && \\
 & && \sum_{j \in I} a_{ij} x_j + \sum_{j \in C} a_{ij} x_j \leq b_i \quad i = 1, \dots, m \\
 & && x_j \in \mathbb{Z}_+ \quad \forall j \in I \\
 & && x_j \in \mathbb{R}_+ \quad \forall j \in C
 \end{aligned}$$

donde I es el conjunto de variables enteras, C es el conjunto de variables continuas y m es la cantidad de restricciones.

Generalmente, hay diferentes formas de representar matemáticamente el mismo problema. De la formulación utilizada puede depender el éxito de resolver en forma óptima grandes instancias en una cantidad de tiempo razonable. Algunas veces, contrariamente a la intuición, puede resultar ventajoso incrementar, en lugar de disminuir, el número de variables o restricciones.

Cada formulación PEM tiene asociado un poliedro $P = \{x \in \mathbb{R}_+^n : Ax \leq b\}$ con $A \in \mathbb{R}^{m \times n}$ y $b \in \mathbb{R}^m$ y el conjunto de soluciones factibles $S = P \cap \{x \in \mathbb{R}^n : x_j \in \mathbb{Z} \forall j \in I\}$. A P se lo denomina relajación lineal de S .

Si llamamos $\text{conv}(S)$ a la cápsula convexa de S (menor poliedro que contiene a S), entonces PEM es equivalente a resolver $\text{Min } cx : x \in \text{conv}(S)$. Si $P = \text{conv}(S)$, el problema PEM puede ser resuelto en forma eficiente por cualquier algoritmo de programación lineal. Este es el caso del conocido *Problema de Transporte*.

Si conociéramos la descripción de $\text{conv}(S)$ mediante un número polinomial (en la cantidad de variables) de desigualdades lineales, podríamos resolver el problema como uno de programación lineal, lo cual es computacionalmente fácil. Es más, aún en el caso que esta caracterización no fuese polinomial, bajo ciertas circunstancias el problema podría ser resuelto en tiempo polinomial. Desafortunadamente, para la mayoría de los problemas no se ha podido obtener la descripción completa de la cápsula convexa y, en general, el número de restricciones lineales que la caracterizan es exponencial.

El procedimiento más simple para resolver un problema de programación entera pura es enumerar todas las posibilidades. Sin embargo, debido a la explosión combinatoria esta técnica sólo resulta aplicable a instancias sumamente pequeñas. En la siguiente sección describimos los algoritmos más usados en la práctica.

2.3. Algoritmos para Problemas de Programación Lineal Entera Mixta

Los algoritmos más utilizados se encuadran dentro de algunos de estos esquemas básicos:

- Enumeración inteligente: algoritmos *Branch-and-Bound*.
- Caracterización de $\text{conv}(S)$ o ajuste de la relajación lineal: algoritmos de planos de corte.
- Una combinación de las dos técnicas anteriores: algoritmos *Branch-and-Cut*.

A continuación describiremos los puntos más sobresalientes de cada uno.

2.3.1. Algoritmos *Branch-and-Bound*

Ya mencionamos que la enumeración de las soluciones factibles en busca de la solución óptima no es un procedimiento aconsejable para usar en la práctica. Para mejorar esta técnica básica muchas veces es posible eliminar algunas posibilidades mediante argumentos de dominancia o factibilidad. Es decir, argumentos que permiten afirmar que el óptimo no pertenece a un determinado subconjunto de soluciones sin la necesidad de enumerarlo.

Dentro de esta línea, en los años 60 fueron propuestos los algoritmos *Branch-and-Bound*, donde el *branching* se refiere a la parte enumerativa y el *bounding* al proceso de poda de posibles soluciones.

Estos algoritmos están asociados al concepto *divide y conquista*: si resulta difícil buscar el óptimo en un conjunto S , entonces es mejor buscar en partes de S y luego quedarse con la mejor solución.

Este esquema puede ser representado mediante un árbol cuya raíz corresponde al problema original y sus ramas resultan de la división en partes del espacio de búsqueda. A cada nodo del árbol le corresponde un subproblema que consiste en buscar el óptimo en una parte del espacio de soluciones. Los argumentos de dominancia y factibilidad son los que permitirán descartar ramas del árbol en el proceso de búsqueda.

Una forma de llevar a cabo la poda, *bounding*, es calcular en los nodos del árbol cotas inferiores del óptimo del problema restringido a esa parte del espacio de soluciones. Si la

cota es peor que la mejor solución obtenida hasta el momento, no es necesario explorar dicha parte. El cálculo de estas cotas debe lograr un equilibrio entre calidad y esfuerzo en obtenerla. Una cota débil hará que se explore innecesariamente ramas del árbol, pero un procedimiento que brinde buenas cotas a un costo alto puede no justificarse.

Para obtener cotas inferiores, una posibilidad es relajar el problema de forma de obtener una relajación *fácil* de resolver. La idea es reemplazar un PEM difícil por un problema de optimización más simple cuyo valor óptimo sea menor o igual al óptimo del problema original. Obviamente, es deseable obtener relajaciones *ajustadas*, es decir, que la diferencia relativa (*gap*) entre el valor óptimo de la relajación y el valor óptimo del PEM sea chica. Hay dos posibilidades obvias para que el problema relajado tenga esta característica. Se puede agrandar el conjunto de soluciones factibles sobre el cual se optimiza o reemplazar la función objetivo por otra que tenga igual o menor óptimo. Dentro de la primera posibilidad se encuentra la relajación lineal y en la segunda se enmarca la relajación lagrangeana. Las relajaciones no sólo son útiles para obtener cotas inferiores, algunas veces permiten probar optimalidad.

La relajación lineal consiste en borrar del PEM la imposición de ser entera sobre las variables que la tengan. Es la relajación más natural y una de las más utilizadas. La relajación lagrangeana consiste en remover un subconjunto de las restricciones que no incluya las restricciones de no negatividad. La violación de las restricciones relajadas es penalizada incluyendo estas restricciones, con un multiplicador no negativo, en la función objetivo. Los multiplicadores son iterativamente actualizados para maximizar la cota inferior obtenida del problema relajado. En [3], Beasley hace una muy buena reseña de la aplicación de esta técnica.

Esencialmente, hay dos factores decisivos en la implementación de un algoritmo de este tipo: las reglas de *branching* y el esquema de selección del próximo nodo a explorar. No hay una combinación de estos factores que resulte mejor para todos los problemas. Es necesario utilizar criterios basados en una combinación de teoría, sentido común y experimentación.

El proceso de *branching* consiste en dividir la región factible anterior en dos o más regiones factibles más pequeñas. Cada nueva región da origen a un nuevo subproblema o nodo hijo, originado por la adición de una nueva restricción al problema del nodo padre. Un requerimiento esencial es que cada solución entera factible del nodo padre pertenezca a, al menos, uno de los hijos. Estos nuevos subproblemas son agregados a la lista de nodos activos, es decir, aún no explorados. La regla de *branching* más simple consiste en considerar alguna variable entera que tiene valor fraccionario, d , en la solución actual. Se parte al problema en dos hijos, imponiendo en uno de ellos como cota superior de esta variable el valor $\lfloor d \rfloor$ y en el otro como cota inferior $\lceil d \rceil$. Este procedimiento es aplicado recursivamente a cada nodo del árbol.

La próxima decisión que se debe tomar es la selección del siguiente nodo a explorar de

la lista de nodos activos. En la práctica hay varios argumentos contradictorios que pueden ser utilizados. Como sólo es posible podar significativamente el árbol de enumeración si se cuenta con buenas cotas superiores, entonces deberíamos descender lo más pronto posible en el árbol para encontrar rápidamente soluciones factibles. Esto sugiere el uso de una estrategia de *búsqueda en profundidad*. Otra estrategia sugiere elegir el nodo activo con mejor cota (más chica). De esta manera, nunca dividiríamos un nodo con cota inferior mayor que el valor óptimo del problema. Esta estrategia es llamada *mejor cota primero*.

El esquema básico del algoritmo es el siguiente. Llamamos *PEM* el problema entero mixto que queremos resolver, \mathcal{N} al conjunto de subproblemas o nodos del árbol de enumeración activos. Para cada nodo k , $PL(k)$ representa la relajación lineal del PEM asociado a este nodo y Z^k el valor óptimo de $PL(k)$. En Z^* se almacena el valor de la mejor solución obtenida.

1. **Inicialización:**

$$\mathcal{N} = \{\text{PEM}\} \quad Z^* = \infty$$

2. **Elección de próximo nodo:**

Si $\mathcal{N} = \{\}$ el algoritmo termina. Si $Z^* \neq \infty$ entonces es óptimo. Si no, PEM es no factible

Si $\mathcal{N} \neq \{\}$, elegir y borrar un nodo k de \mathcal{N}

3. **Evaluación:**

Resolver $PL(k)$.

a) Si no es factible, ir a **Elección**.

b) Asignar a Z^k el valor óptimo de $PL(k)$.

c) *Bound*: si $Z^k > Z^*$, ir a **Elección**.

d) Si la solución óptima cumple las condiciones de integralidad, actualizar $Z^* = \min\{Z^*, Z^k\}$ e ir a **Elección**.

4. **División:** (*Branch*) Particionar la región factible de $PL(k)$ en dos o más regiones, agregando un nuevo nodo a \mathcal{N} por cada nueva región. Ir a **Elección**.

2.3.2. Algoritmos de Planos de Corte

Los algoritmos de planos de corte fueron originalmente propuestos por Gomory en los '60 [4] como un método general para resolver problemas de programación lineal entera.

Un algoritmo básico de planos de corte en un primer paso relaja las condiciones de integralidad sobre las variables y resuelve el programa lineal resultante. Si el programa lineal es

infactible, el programa entero también lo es. Si la solución óptima del programa lineal cumple las condiciones de integralidad, se ha encontrado un óptimo del problema. En caso contrario, se busca identificar desigualdades lineales (*problema de separación*) que estén violadas por la solución fraccionaria del programa lineal y sean válidas para los puntos enteros factibles. Es decir, desigualdades que *separen* el óptimo fraccionario de $\text{conv}(S)$.

El algoritmo continúa hasta que:

- una solución entera es encontrada, en cuyo caso el problema es resuelto con éxito
- el programa lineal es infactible, lo que significa que el problema entero es infactible
- no se pudo identificar alguna desigualdad lineal, ya sea porque no se conoce la descripción completa de la cápsula convexa o porque estos algoritmos de separación no son exactos.

El éxito del algoritmo depende en gran medida de la posibilidad y la eficiencia de encontrar desigualdades violadas (*planos de corte*) que puedan ser agregadas a la formulación para separar las soluciones fraccionarias.

Los planos de corte pueden ser generados bajo dos enfoques:

- *Con herramientas generales aplicables a cualquier problema de programación lineal entera*

El algoritmo original propuesto por Gomory utiliza como planos de corte desigualdades derivadas del *tableau* óptimo de la relajación lineal, llamados *cortes de Gomory*. Aunque fue demostrado que este algoritmo, bajo ciertas condiciones, termina en un número finito de pasos, en la práctica su convergencia parece ser muy lenta. Por otro lado, la implementación computacional es numéricamente inestable, aunque en la actualidad han sido fortalecidos lográndose buenas implementaciones.

Posteriormente, se han desarrollado algoritmos que utilizan una variedad de cortes aplicables a cualquier *PEM*, como por ejemplo los cortes *disyuntivos*, *clique*, *cover*, etc. Si bien estos algoritmos tienen propiedades teóricas de mucho interés, su éxito en la práctica es discutible. Cualquiera de las técnicas mencionadas tiene la ventaja de poder ser utilizadas para cualquier problema de programación entera, independientemente de su estructura. Si bien esto es una propiedad deseable en un algoritmo, no siempre brinda la herramienta más adecuada para casos particulares. Un estudio más específico del problema ayuda a obtener mejores procedimientos. Este es el sentido del próximo enfoque.

- *Explotando la estructura particular del problema.*

En los '70, resurgió el interés por los algoritmos de planos de corte debido al desarrollo de la teoría poliedral. Mediante el estudio de combinatoria poliedral, la intención es reemplazar el conjunto de restricciones de un programa de programación entera mixta por la descripción de la cápsula convexa del conjunto de soluciones factibles. Las desigualdades lineales necesarias para describir a $\text{conv}(S)$ se denominan *facetas*. Si se conoce de forma completa esta descripción, el problema entero puede ser resuelto como un problema de programación lineal. De esta manera, explotando la estructura particular de cada problema, los planos de corte resultarán más efectivos a la hora de cortar soluciones.

Desafortunadamente, no es fácil tener esta descripción y los problemas pertenecientes a la clase NP-Difícil tienen una cantidad exponencial de facetas, a menos que $P = NP$. Alternativamente, es posible utilizar cualquier desigualdad válida para el conjunto de soluciones factibles como planos de corte, pero, en general, la eficiencia del algoritmo depende de la *fortaleza* de estas desigualdades, siendo las facetas los *mejores* cortes posibles.

Con fines algorítmicos, el estudio poliedral debe estar acompañado de algoritmos de separación eficientes. En este sentido, hay un resultado muy importante debido a Grötschel, Lovász y Schrijver [5] que relaciona la complejidad del problema de separación con la complejidad del problema de optimización. Se establece que el problema de optimización $\max\{cx : x \in \text{conv}(S)\}$ puede resolverse polinomialmente si y sólo si el problema de separación ($x \in \text{conv}(S)$ ó encontrar una desigualdad válida violada) es polinomial. Es decir que si el problema que estamos tratando no es polinomial, existe al menos alguna familia de facetas que no puede separarse en tiempo polinomial. Esto de alguna manera implica el grado de dificultad de encontrar la descripción de todas las facetas de la cápsula convexa y del desarrollo de algoritmos de separación.

En forma general, para desarrollar un algoritmo de planos de corte, primero se busca una descripción parcial de la cápsula convexa del conjunto de las soluciones factibles enteras o desigualdades válidas *fuertes* para este conjunto. Luego es necesario el diseño de rutinas de separación para las familias de desigualdades encontradas. Estas rutinas toman como entrada una solución y retornan restricciones de estas familias violadas por este punto, si es que existe alguna. El problema de separación, en algunos casos, puede ser NP-difícil o tener complejidad alta, lo que lleva en la práctica a utilizar algoritmos heurísticos, o sea, que es posible que no sean capaces de encontrar una desigualdad violada aunque exista. La estrategia que se utilice para decidir la búsqueda en la diferentes familias es clave para la performance del algoritmo.

El esquema básico de un algoritmo de planos de corte es el siguiente. Llamamos *PEM* al problema entero mixto que queremos resolver, $PL(P)$ a la relajación lineal del problema

P y x^P la solución óptima de esta relajación.

1. **Inicialización:**

$P = PEM$

2. **Evaluación:**

Resolver $PL(P)$

- a) Si es no factible, entonces PEM es no factible y el algoritmo termina.
- b) Si x^P cumple las condiciones de integralidad, x^P es la solución óptima de PEM y el algoritmo termina.
- c) **Separación:** Caso contrario, resolver el problema de separación para x^P .
 - Si se encuentran cortes, agregarlos a P e ir a **Evaluación**.
 - Caso contrario, retornar el funcional evaluado en x^P como una cota inferior de PEM .

El algoritmo de planos de corte puede no resolver el problema de forma óptima, ya sea por no encontrar desigualdades válidas violadas o porque el tiempo consumido excede el tiempo disponible. Sin embargo, puede ser utilizado para generar buenas cotas inferiores del valor óptimo del problema. Además, muchas veces a partir de la solución óptima de la relajación actual es posible encontrar buenas soluciones enteras mediante una heurística, brindando una cota superior del valor óptimo.

2.3.3. Algoritmos Branch-and-Cut

En muchas instancias, los dos algoritmos descriptos arriba fallan en la resolución del problema. A comienzos de los '80 se comenzó a aplicar una metodología mixta que conjuga las dos ideas dando origen a los llamados algoritmos *Branch-and-Cut*. De esta manera se lograron resolver exitosamente instancias de tamaño considerable de una gran cantidad de problemas de programación lineal entera, como por ejemplo el *Problema de Viajante de Comercio*, el *Problema de Ordenamiento Lineal*, el *Problema de Corte Máximo*, etc.

Uno de los factores que influye en el fracaso de los algoritmos *Branch-and-Bound* es la baja calidad de las cotas obtenidas mediante las relajaciones lineales. Esto significa que resulta crucial poder ajustar las formulaciones, por ejemplo con planos de corte.

Un algoritmo *Branch-and-Cut* es un *Branch-and-Bound* en el cual se generan planos de corte a través del árbol de enumeración. El objetivo de esto es reducir significativamente el

número de nodos del árbol mejorando la formulación de los subproblemas. En un *Branch-and-Cut*, la enumeración y los planos de corte se benefician mutuamente. Generalmente, la cota producida en cada nodo del árbol de enumeración es mejor que en un *Branch-and-Bound*, debido a las nuevas desigualdades agregadas a la formulación del correspondiente subproblema. Por otro lado, el proceso de *branching* perturba la solución fraccionaria ayudando a los algoritmos de separación.

Estos algoritmos no sólo son capaces de producir la solución óptima, también pueden brindar soluciones aproximadas al óptimo con certificado de calidad en tiempos de cómputo moderado.

En la implementación de un algoritmo *Branch-and-Cut* hay que tener en cuenta las estrategias propias de un algoritmo *Branch-and-Bound* sumadas a las de un algoritmo de planos de corte. Además, se agregan nuevas decisiones como ¿cuándo buscar planos de cortes?, ¿cuántos cortes agregar?, etc.

El esquema de un algoritmo *Branch-and-Cut* es el siguiente.

1. **Inicialización:**

$$\mathcal{N} = \{\text{PEM}\} \quad Z^* = \infty$$

2. **Elección de próximo nodo:**

Si $\mathcal{N} = \{\}$ Z^* es óptimo y el algoritmo termina
Si no, elegir y borrar un nodo k de \mathcal{N}

3. **Evaluación:**

Resolver $PL(k)$.

a) Si es no factible, ir a **Elección**.

b) Asignar a Z^k el valor óptimo de $PL(k)$.

c) Si $Z^k > Z^*$, ir a **Elección**.

d) Si la solución óptima cumple las condiciones de integralidad, poner $Z^* = \min\{Z^*, Z^k\}$ e ir a **Elección**.

4. **División vs Separación:**

Decidir si se buscarán planos de corte:

- **SI:** Ir a **Separación**
- **NO:** Ir a **División**

5. **División:** Particionar la región factible de $PL(k)$ en dos o más regiones, agregando un nuevo nodo a \mathcal{N} por cada nueva región. Ir a **Elección**.

6. **Separación:** Resolver el problema de separación para la solución fraccionaria de $PL(k)$.

- Si son encontrados cortes, agregarlos a la formulación e ir a **Evaluación**.
 - Si no se encuentran, ir a **División**.
-

Capítulo 3

Modelo de Programación Lineal Entera para el PV

En esta sección se presenta el Problema de los Visitadores como problema de Programación Lineal Entera. Los comercios 0 y $C + 1$ representan la casa del visitador. Si bien ambos tienen el mismo significado, utilizamos esta distinción para facilitar la representación del modelo.

Definimos D_v como la distancia total del recorrido del visitador v . Además utilizamos d_{c_1, c_2} para denotar la distancia existente entre los comercios c_1 y c_2 .

3.1. Modelo

Definimos las siguientes variables:

$$Z_{v, c_1, c_2} = \begin{cases} 1 & \text{si el visitador } v \text{ visita el comercio } c_2 \text{ justo después del comercio } c_1 \\ 0 & \text{en caso contrario} \end{cases}$$

$$\forall v \in \{1, \dots, V\}, \forall c_1 \in \{1, \dots, C\}, \forall c_2 \in \{1, \dots, C\}, c_1 \neq c_2$$

$$Z_{v, 0, c} = \begin{cases} 1 & \text{si el visitador } v \text{ visita como primer comercio a } c \\ 0 & \text{en caso contrario} \end{cases}$$

$$\forall v \in \{1, \dots, V\}, \forall c \in \{1, \dots, C\}$$

$$Z_{v, c, C+1} = \begin{cases} 1 & \text{si el visitador } v \text{ visita como último comercio a } c \\ 0 & \text{en caso contrario} \end{cases}$$

$$\forall v \in \{1, \dots, V\}, \forall c \in \{1, \dots, C\}$$

$$\min \sum_{v=1}^V D_v$$

s.a.

$$D_v = \sum_{c_1=1, c_1 \neq c_2=1}^C \sum_{c_2}^C d_{c_1, c_2} Z_{v, c_1, c_2} \quad \forall v \in \{1, \dots, V\} \quad (3.1)$$

$$\sum_{c_1=1, c_1 \neq c_2}^C \sum_{c_2=1}^C Z_{v_1, c_1, c_2} - \sum_{c_1=1, c_1 \neq c_2}^C \sum_{c_2=1}^C Z_{v_2, c_1, c_2} \leq 1 \quad \forall v_1, v_2 \in \{1, \dots, V\}, v_1 \neq v_2 \quad (3.2)$$

$$\sum_{c=1}^C Z_{v, 0, c} = 1 \quad \forall v \in \{1, \dots, V\} \quad (3.3)$$

$$\sum_{c=1}^C Z_{v, c, C+1} = 1 \quad \forall v \in \{1, \dots, V\} \quad (3.4)$$

$$\sum_{v=1}^V \sum_{c_1=0, c_1 \neq c_2}^C Z_{v, c_1, c_2} = 1 \quad \forall c_2 \in \{1, \dots, C\} \quad (3.5)$$

$$\sum_{v=1}^V \sum_{c_2=1, c_2 \neq c_1}^{C+1} Z_{v, c_1, c_2} = 1 \quad \forall c_1 \in \{1, \dots, C\} \quad (3.6)$$

El objetivo del problema es minimizar la suma de las distancias de los recorridos de todos los visitantes, definiendo mediante las ecuaciones (3.1) esta distancia.

Las restricciones (3.2), **homogeneidad de cantidad de comercios (HCC)**, imponen que todos los visitantes tengan la misma cantidad de comercios en su recorrido o a lo sumo sus cantidades difieran en 1. Estas dos familias de restricciones sirven para asegurarnos que lo siguiente se cumpla:

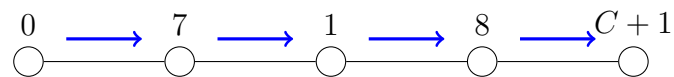
$$\left| \sum_{c_1=1, c_1 \neq c_2}^C \sum_{c_2=1}^C Z_{v_2, c_1, c_2} - \sum_{c_1=1, c_1 \neq c_2}^C \sum_{c_2=1}^C Z_{v_1, c_1, c_2} \right| \leq 1 \quad \forall v_1, v_2 \in \{1, \dots, V\}, v_1 \neq v_2$$

Las restricciones de **unicidad inicial (UI)**, (3.3) obligan a que todo visitador tenga un y sólo un comercio inicial al cual va desde su casa, y las de **unicidad final (UF)**, (3.4), a que todo visitador tenga un y sólo un comercio final desde el cual va a su casa.

Las restricciones (3.5), de **unicidad de origen (UO)** imponen que a todo comercio llegue un y sólo un visitador desde un único origen (otro comercio o su casa) y las (3.6), de **unicidad de destino (UD)** que de todo comercio parta un y sólo un visitador que va a un único destino (otro comercio o su casa).

Con las restricciones mencionadas hasta este momento aún puede darse la existencia de soluciones no válidas para nuestro problema. A continuación se detalla el análisis de los posibles casos:

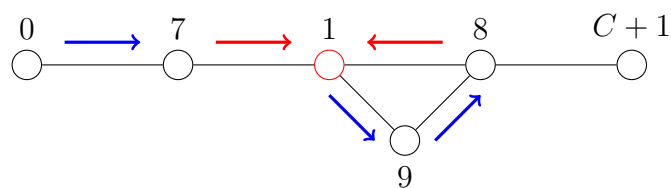
1. Camino sin ciclos:



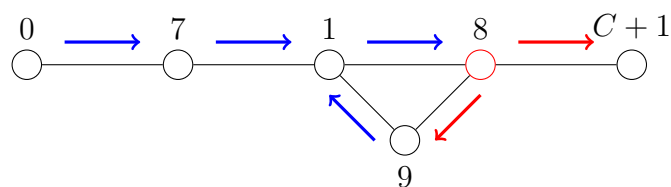
Este tipo de camino es un recorrido válido para el modelo. Esto es correcto porque es el tipo de caminos que queremos modelar.

2. Camino con ciclos: El modelo debe evitar este tipo de caminos.

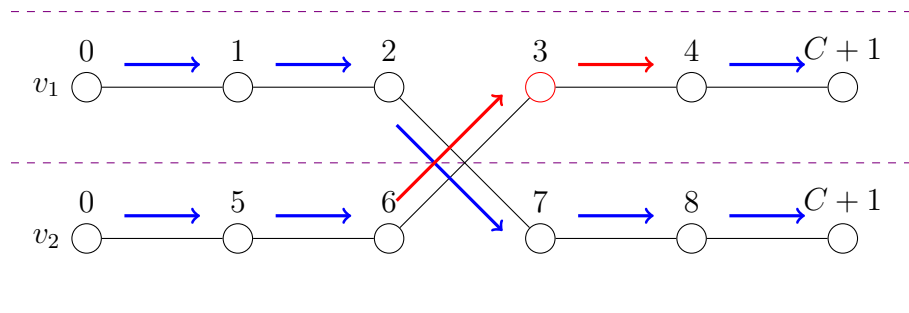
- Este caso no es válido para el modelo ya que no cumple con la restricción de unicidad de origen (UO).



- Este caso no es válido para el modelo ya que no cumple con la restricción de unicidad de destino (UD).



3. Caminos cruzados:

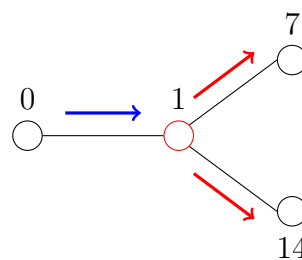


Claramente este tipo de caminos no es válido en las soluciones factibles del problema PV, ya que el visitador 1 visita el comercio 4 después del comercio 3 y además el visitador 2 visita el comercio 6 antes del comercio 3. Es decir, que el comercio 3 es visitado por ambos visitadores. Pero desafortunadamente el modelo propuesto con las restricciones anteriores lo permite.

Para corregir el modelo teniendo en cuenta este inconveniente encontramos planteamos una nueva familia de restricciones llamada restricciones de **entrada y salida (EYS)**, que establecen que a todo comercio llegue y salga el mismo visitador.

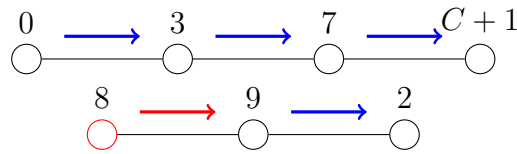
$$\sum_{c_1=0}^C Z_{vc_1c_2} = \sum_{c_1=1}^{C+1} Z_{vc_2c_1} \quad \forall v \in \{1, \dots, V\}, \forall c_2 \in \{1, \dots, C\}$$

4. Árbol (con bifurcación de caminos): Este caso no es válido para el modelo ya que no cumple con la restricción de unicidad de destino (UD).

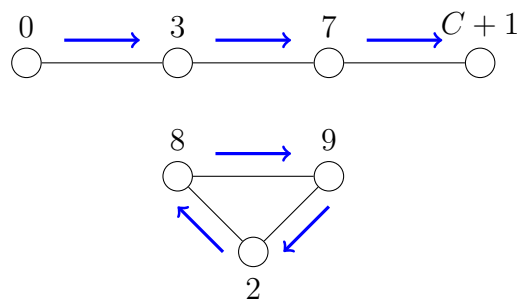


5. Recorrido partido de un visitador (grafo con más de una componente):

- Este caso no es válido para el modelo ya que no cumple con la restricción de entrada y salida (EYS). Esto puede verse ya que la variable $Z_{v,8,9} = 1$ y todas las variables $Z_{v,x,8} = 0$ (donde x es cualquier valor entre 0 y C). Con lo cual la igualdad de la restricción no se cumple.



- Claramente se ve que este caso no debería ser válido para el modelo ya que el visitador tiene el recorrido partido. Pero desafortunadamente el modelo propuesto con las restricciones actuales lo permite.



Nuevamente, para corregir el modelo teniendo en cuenta este inconveniente, incorporamos la familia de restricciones de **eliminación de subtour (EST)**.

$$\sum_{c_1 \in S} \sum_{c_2 \in S} Z_{vc_1c_2} \leq |S| - 1 \quad \forall v \in \{1, \dots, V\}, \forall S \subseteq \{1, \dots, C\}$$

Si hay un ciclo en el recorrido la desigualdad no se cumple para el caso que S esté formado exclusivamente por los comercios del ciclo.

Notar que la cantidad de restricciones de esta familia crece exponencialmente. En la sección “4.1.Detección de Ciclos” describiremos cómo manejamos esta característica.

3.2. Análisis de la función objetivo

Luego de analizar detalladamente el problema notamos que el objetivo planteado permitía obtener distancias de recorridos de visitantes muy dispares. Si dejáramos el objetivo así planteado, las soluciones que se obtendrían perjudicarían a aquellos vendedores con recorridos más largos.

Para eliminar esta característica, para todo par de visitantes $v_1, v_2 \in \{1, \dots, V\}, v_1 < v_2$, agregamos la variable continuas t_{v_1, v_2} que representan la diferencia entre la distancia total

recorrida por el visitador v_1 y la distancia total recorrida por el visitador v_2 y replanteamos la función objetivo obteniendo la siguiente función multiobjetivo:

$$\text{mín } \alpha \sum_{v=1}^V D_v + \beta \sum_{v_1=1}^{V-1} \sum_{v_2=v_1+1}^V t_{v_1, v_2}$$

Para definir el valor de t_{v_1, v_2} definimos las restricciones de **diferencias de distancias entre visitadores (DDV)** $\forall v_1, v_2 \in \{1, \dots, V\}, v_1 < v_2$:

$$\begin{aligned} t_{v_1 v_2} &\geq D_{v_1} - D_{v_2} \\ t_{v_1 v_2} &\geq D_{v_2} - D_{v_1} \end{aligned}$$

Con estas restricciones nos aseguramos que t_{v_1, v_2} sea mayor o igual al módulo de la diferencia de la distancia total recorrida por el visitador v_1 y por el visitador v_2 . Esto, en conjunto con la minimización de la función objetivo, garantiza que t_{v_1, v_2} sea exactamente igual al módulo de la diferencia.

Los parámetros α y β sirven para poder darle más importancia a uno u otro objetivo de la función. Cabe aclarar que estos valores serán parámetros de entrada de la implementación del problema y deberán ser valores enteros.

Capítulo 4

Implementación del algoritmo Branch-and-Cut

En este capítulo describiremos las características principales de nuestro algoritmo *Branch-and-Cut* al cual llamaremos *BC-V*. Como mencionamos anteriormente, la performance de un algoritmo *Branch-and-Cut* depende de muchos factores. Una buena relajación lineal y planos de corte con procedimientos de separación adecuados, ayuda al mejoramiento de las cotas inferiores. También evaluamos heurísticas para obtener cotas primales y otros detalles de implementación. Esto es clave para podar ramas del árbol y disminuir el espacio de búsqueda. No hay una elección óptima de cada una de las alternativas del algoritmo para cualquier problema ni instancia. Las características propias del problema son las que ayudan a determinar una buena elección.

Las instancias utilizadas para estudiar el comportamiento del algoritmo y sus variantes han sido generadas creando puntos al azar en un cuadrado de lado 20 el cual representa la zona donde se distribuirán los comercios. Para esta generación de puntos no se utilizó ninguna restricción adicional.

Se estudiaron diversas familias presentando en las tablas los promedios para 6 instancias en cada caso. Los nombres de las familias tienen el formato vX_cY donde X es la cantidad de visitantes e Y es la cantidad de comercios. Por ejemplo, la familia $v4_c18$ tiene 4 visitantes y 18 comercios. Las familias utilizadas variaron en tamaño desde la $v2_c7$ hasta la $v5_c21$.

4.1. Detección de ciclos

Llegado el momento de implementar un algoritmo *Branch-and-Cut* en base al modelo planteado, surgió el problema de que la cantidad de restricciones de eliminación de subtour crece exponencialmente, impactando considerablemente en la performance del algoritmo. La resolución de estas relajaciones lineales consumen una cantidad de tiempo prohibitiva den-

tro del contexto de un algoritmo *Branch-and-Cut*. Dado este problema decidimos relajar el modelo no incluyendo estas restricciones, agregándolas a medida que se vaya detectando que las mismas han sido violadas. Estas restricciones serán agregadas como planos de cortes si se obtiene una solución entera una vez ejecutada la optimización del problema relajado linealmente.

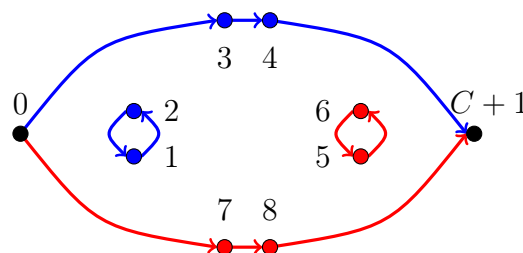
A la hora de implementar la verificación de la validez de las restricciones de subtour, el primer enfoque estudiado fue a través de fuerza bruta. Es decir, tomar cada posible conjunto de comercios que pueda formarse con los que están en el recorrido de cada visitador y chequear si se forma un ciclo. En caso de detectar un ciclo, se agrega la restricción de eliminación de subtour correspondiente para el vendedor que tiene el ciclo en su recorrido. Las implementaciones por fuerza bruta no son eficientes si el tamaño del problema es mediano o grande, pero sirve como primera aproximación en la búsqueda de un algoritmo que resuelva el problema de los visitadores eficientemente. Llamaremos a esta versión **FB1**.

Las mediciones de performance realizadas ejecutando un Branch-and-Bound con esta versión son las siguientes:

Caso	V	C	#Instancias	FB1
v2_c7	2	7	6	0.73 seg
v2_c10	2	10	6	2.66 min
v3_c10	3	10	6	1.41 hs
v3_c12	3	12	6	5.26 hs

Tabla 4.1: Desempeño FB1

Luego de evaluar los resultados anteriores encontramos una optimización para la versión FB1. Veamos el siguiente ejemplo que tiene 8 comercios y 2 visitadores para recorrerlos:



Una posible primera solución que puede obtener el algoritmo es que el visitador A (color azul) tenga en su recorrido los comercios 1, 2, 3 y 4 y que el visitador B (color rojo) tenga los comercios 5, 6, 7 y 8. Los comercios 1, 2 formando un ciclo y los comercios 5 y 6 también. En FB1 se detectan estos dos ciclos y se agregan las siguientes restricciones al problema:

- No puede haber un ciclo formado por los comercios 1 y 2 **en el recorrido del visitador A.**
- No puede haber un ciclo formado por los comercios 5 y 6 **en el recorrido del visitador B.**

En una futura iteración el algoritmo podría formar alguno de estos mismos ciclos pero intercambiando los visitadores. Esto se debe a que los visitadores son indistinguibles y resulta indiferente a quien se le asigna cada recorrido.

Esta característica se analizará más adelante en la sección “4.5.Rompimiento de Simetría”.

4.1.1. Variantes FB2 y FB3

En base a esta situación, realizamos una optimización que consiste en agregar las restricciones de eliminación de subtour encontradas **a todos los visitadores** y no solamente al visitador que tenía los comercios involucrados en su recorrido. Llamaremos a esta versión **FB2**.

Otra variación realizada fue limitar la cantidad de restricciones que se agregan al problema en cada iteración. En FB2 se agregan todas las restricciones posibles en cada iteración y se optimiza. Con esta variación agregamos menos restricciones por iteración con el fin de que cada iteración insuma menos tiempo. La cantidad máxima de restricciones que permitimos agregar por iteración depende del tamaño del problema (cantidad de visitadores y cantidad de comercios). Dicha variación logra reducir el tiempo por iteración pero generalmente aumenta la cantidad de iteraciones que se realizan. Llamaremos a esta versión **FB3**.

La forma para determinar la cantidad máxima de restricciones que permitimos agregar por iteración se calcula con la siguiente formula:

$$R_{max} = F \times V \times C^2$$

donde V es la cantidad de visitadores, C la cantidad de comercios y F es un factor utilizado para variar la cantidad de las restricciones totales agregadas. A continuación se muestra una comparación de los tiempos obtenidos utilizando tres distintos valores para F , los cuales determinamos empíricamente.

Dados los resultados observados, utilizaremos la variante FB3 con el factor F establecido en 0.015.

Caso	Visitadores	Comercios	#Inst.	F = 0.015	F = 0.03	F = 0.05
v2_c7	2	7	6	1.35 seg	1.47 seg	1.52 seg
v2_c10	2	10	6	2.9 min	3.01 min	3.23 min
v3_c10	3	10	6	1.05 hs	1.15 hs	1.34 hs
v3_c12	3	12	6	5.66 hs	6.09 hs	7.54 hs

Tabla 4.2: Análisis factor F en FB3

Comparación de desempeño

En la siguiente tabla se muestra la comparación entre los resultados obtenidos al resolver las mismas instancias con **FB1**, **FB2** y **FB3**:

Caso	Visitadores	Comercios	#Instancias	FB1	FB2	FB3
v2_c7	2	7	6	0.73 seg	0.77 seg	1.35 seg
v2_c10	2	10	6	2.66 min	2.81 min	2.9 min
v3_c10	3	10	6	1.12 hs	0.93 hs	1.05 hs
v3_c12	3	12	6	5.26 hs	5.61 hs	5.66 hs

Tabla 4.3: Desempeño FB1, FB2 y FB3

La mejora de performance que suponíamos que la versión FB2 iba a ofrecer sobre FB1 no se cumplió. En la mayoría de los casos los tiempos no mejoraron sino que por el contrario empeoraron un poco. Suponemos que esto se debe a que es más costoso iterar menos veces con más restricciones en cada iteración que iterar más veces con menos restricciones.

Siguiendo con la suposición de que era mejor iterar menos veces con la mayor cantidad de restricciones y tratando de confirmar que esto era cierto, implementamos FB3 suponiendo que los tiempos resultantes se ubicarían entre los tiempos de FB1 y FB2. Esto tampoco se cumplió.

Luego de la comparación de las tres implementaciones, concluimos que es mejor tener más iteraciones pero que las mismas no sean demasiado costosas. Esta conclusión se basa en la experiencia obtenida hasta el momento con los tamaños de instancias utilizados.

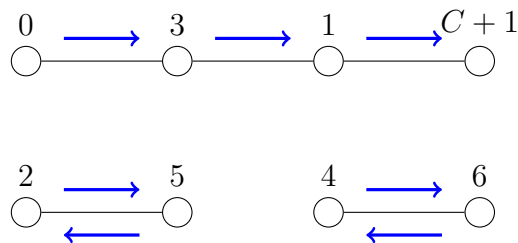
4.1.2. Variante de optimización de subtour

Con el fin de no introducir restricciones redundantes realizamos un análisis profundo de las implementaciones anteriores manteniendo la idea de optimizar la cantidad de restricciones que agregamos por iteración. La finalidad de esto último es tener iteraciones cuyos tiempos

de ejecución sean lo más acotados posibles. Tener en cuenta que en general, cuántas más restricciones se agregan en una iteración, más tiempo se necesita para calcular la solución.

Durante este análisis detectamos la siguiente particularidad. Cuando en el camino de un visitador detectamos dos ciclos, agregamos una restricción para cada ciclo y además agregamos una restricción adicional que involucra a los comercios incluidos en ambos ciclos y solamente a esos.

Para clarificar veamos el siguiente ejemplo. El visitador 1 tiene su camino compuesto por los comercios 1, 2, 3, 4, 5 y 6. Los comercios 2 y 5 forman un ciclo y los comercios 4 y 6 forman otro.



Con lo cual el algoritmo agrega las siguientes restricciones EST (ver modelo):

$$R_1: Z_{1,2,5} + Z_{1,5,2} \leq |S| - 1; S = \{2, 5\}.$$

$$R_2: Z_{1,4,6} + Z_{1,6,4} \leq |S| - 1; S = \{4, 6\}.$$

$$R_3: Z_{1,2,4} + Z_{1,2,5} + Z_{1,2,6} + Z_{1,4,2} + Z_{1,4,5} + Z_{1,4,6} + Z_{1,5,2} + Z_{1,5,4} + Z_{1,5,6} + Z_{1,6,2} + Z_{1,6,4} + Z_{1,6,5} \leq |S| - 1; S = \{2, 4, 5, 6\}.$$

Podemos ver que para esta solución el lado izquierdo de R_3 se reduce a $Z_{1,2,5} + Z_{1,4,6} + Z_{1,5,2} + Z_{1,6,4}$ dado que las otras variables valen 0. De esta manera se puede ver que si se cumplen las restricciones R_1 y R_2 , también se cumple R_3 . Con lo cual podemos suponer que no se obtendrá una mejora al agregar R_3 al problema y planteamos considerar agregar solo restricciones EST sobre ciclos minimales.

Por último, es importante remarcar que cuántos más ciclos tenga el camino de un visitador, la cantidad de restricciones sobre ciclos no minimales agregadas crece exponencialmente. Sea n la cantidad de ciclos existentes en el recorrido de un visitador, la cantidad de restricciones sobre ciclos no minimales que se agregan es $\sum_{i=2}^n \binom{n}{i}$. Por ejemplo, si el camino contiene 3 ciclos, se agregarían 4 restricciones sobre ciclos no minimales, si el camino contiene 4 ciclos, se agregarían 11 restricciones sobre ciclos no minimales y si el camino contiene 5 ciclos, se agregarían 26 restricciones sobre ciclos no minimales. Llamaremos a esta versión **OST** (Optimización de Subtour).

Comparación de desempeño

A continuación se muestra la comparación entre los resultados obtenidos al resolver las mismas instancias sin OST y con OST (ambas variantes se han ejecutado utilizando FB1):

Caso	Visitadores	Comercios	#Instancias	Sin OST	Con OST
v2_c9	2	9	6	24.18 seg	24.61 seg
v2_c10	2	10	6	2.66 min	2.91 min
v2_c12	2	12	6	20.7 min	22.9 min
v2_c14	2	14	6	85.19 min	48.44 min
v2_c17	2	17	6	1.94 hs	1.30 hs

Tabla 4.4: Desempeño Optimización de Subtour

Analizando los resultados anteriores pudimos confirmar que la mejora de performance que suponíamos efectivamente se cumple, aunque a partir de instancias de un cierto tamaño.

En las instancias de la tabla que tiene hasta 12 comercios la cantidad de ciclos máxima que pueden formarse en el recorrido de un visitador es de 2 (la cantidad mínima de comercios necesarios para formar al menos 3 ciclos es de 7 comercios ya que uno siempre debe quedar suelto para cumplir con las restricciones del modelo). Recordar que cuantos más ciclos haya en el recorrido de un visitador, más restricciones sobre ciclos no minimales se agregan al modelo al no utilizar la optimización OST.

En estos primeros tres casos se agregan como máximo en cada iteración 2 restricciones sobre ciclos no minimales (una por cada visitador). En el caso de 14 comercios se agregan como máximo en cada iteración 8 restricciones sobre ciclos no minimales y en el caso de 17 la cantidad máxima es de 22 restricciones.

4.2. Heurística Inicial (HI)

La experiencia demuestra que contar con cotas superiores iniciales de buena calidad es fundamental para reducir el tamaño del árbol de búsqueda en un algoritmo *Branch-and-Cut*. La eficiencia del proceso de *Bound* (poda) depende en gran medida de la bondad de las cotas. Sin embargo, en un contexto de un algoritmo *Branch-and-Cut*, es necesario lograr un equilibrio entre el tiempo requerido para obtener las cotas y la calidad de las mismas. Por este motivo elegimos procedimientos rápidos y sencillos que, de acuerdo a nuestra experimentación, brindan cotas de muy buena calidad.

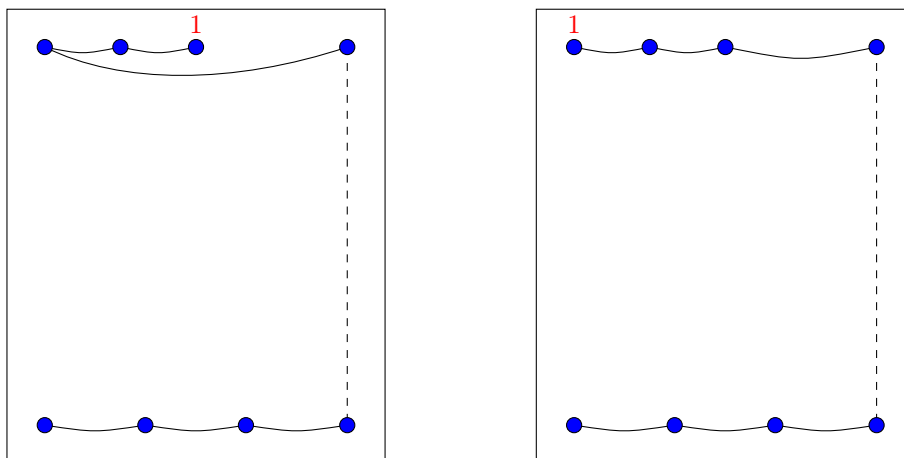
Siguiendo con esta idea, desarrollamos una heurística inicial que se ejecuta una única vez al inicio del proceso y tiene como objetivo brindar una buena cota superior inicial a bajo costo.

La idea de nuestra heurística inicial es la siguiente. Para armar el recorrido del primer visitador se toma un comercio como inicial y se lo asigna como primer comercio de su recorrido. A partir de este se busca el comercio más cercano para ser asignado como segundo comercio. Así sucesivamente se van asignando los restantes comercios al recorrido del visitador respetando la restricción **HCC** (homogeneidad de cantidad de comercios). Luego se repite la lógica descripta para cada visitador hasta completar los recorridos de todos.

En la primera versión implementada se toma como primer comercio del recorrido del primer visitador al comercio etiquetado con 1. Luego de realizar diversas pruebas con dicha implementación, notamos que en algunos casos la cota inicial obtenida no era buena ya que, dado el tamaño de los casos, uno podía ver “a simple vista” una mejor cota con simplemente comenzar la heurística con otro comercio inicial.

A partir de esta observación decidimos modificar la implementación de nuestra heurística de la siguiente manera. Se realiza la lógica anteriormente descripta C veces (cantidad de comercios) tomando en cada oportunidad al comercio c_i como comercio inicial. De las C cotas obtenidas seleccionamos la mejor de todas para ser utilizada por nuestro algoritmo *Branch and Cut*.

A continuación se puede observar cómo varía el valor de la cota obtenida por la heurística dependiendo del comercio inicial. El ejemplo utilizado es de 8 comercios y 2 visitadores y la línea punteada representa la selección del primer comercio del recorrido del segundo visitador a partir del último comercio del recorrido del primer visitador.



4.2.1. Pseudocódigo de la Heurística Inicial

V es la cantidad de visitantes

C es la cantidad de comercios

CI es el comercio inicial

CA es el comercio actual

CC es la cantidad de comercios sin asignar en el recorrido del visitador actual

VA es el visitador actual

MS es el mejor valor de la función objetivo obtenido hasta el momento

SA es el valor de la función objetivo de la solución actual

```

se almacena en  $MS$  el valor infinito
para cada posible comercio inicial  $CI$  hacer
  marcar todos los comercios como sin asignar
  asignar a  $CA$  el valor de  $CI$ 
  asignar  $CA$  como primer comercio del visitador 1
  marcar  $CA$  como comercio asignado
  para  $VA$  desde 1 a  $V$  hacer
     $CC = (C/V - 1)$  si  $VA = 1$  o  $(C/V)$  si  $VA > 1$ 
    hacer  $CC$  veces
      buscar el comercio más cercano a  $CA$  entre los aún no asignados
      asignar a  $CA$  este comercio
      asignar  $CA$  como siguiente comercio de  $VA$ 
      marcar  $CA$  como comercio asignado
  asignar a  $SA$  el valor de la función objetivo en la solución actual
  asignar a  $MS$  el mejor valor entre  $SA$  y  $MS$ 

```

Comparación de desempeño

Usando HI generalmente los tiempos de ejecución mejoran dado que la HI provee una solución cuyo valor objetivo es una cota superior inicial y ésta permite realizar podas sobre el árbol de búsqueda sin la necesidad de tener que esperar a obtener una solución entera de la relajación. De todos modos realizamos las pruebas correspondientes para nuestro algoritmo y confirmamos que efectivamente los tiempos mejoraron.

4.3. Variables X y V

Hasta el momento estuvimos utilizando el “branching” por defecto que usa Cplex asignando las mismas prioridades para todas las variables (Z y T). Con la idea de poder mejorar esto y teniendo en mente la indistinguibilidad de los visitantes ya mencionada, definimos las siguientes variables a incluir en la implementación:

Variables X : En una solución entera la variable X_{ij} indica si algún visitante visita al comercio j inmediatamente después de visitar el comercio i .

$$X_{ij} = \begin{cases} 1 & \text{si un visitante visita } j \text{ inmediatamente después de visitar } i \\ 0 & \text{en caso contrario} \end{cases}$$

Definimos

$$X_{ij} = \sum_{v=1}^V Z_{v,i,j} \quad \forall i, j \in \{1, \dots, C\}, i \neq j$$

Asignar a estas variables una mayor prioridad que las variables Z en el *branching* mejora el desempeño del algoritmo ya que disminuye la cantidad de nodos que se generan en el árbol de búsqueda haciéndolo más balanceado.

Variables V : En una solución entera la variable V_{vc} indica si el comercio c es visitado o no por el visitante v , sin importar el orden dentro del recorrido.

$$V_{vc} = \begin{cases} 1 & \text{si } v \text{ visita a } c \\ 0 & \text{en caso contrario} \end{cases}$$

Definimos

$$V_{vc} = \sum_{j=1}^{C+1} Z_{v,c,j} \quad \forall v \in \{1, \dots, V\}, \forall c \in \{1, \dots, C\}$$

Del mismo modo que ocurre con las variables X , asignar a las variables V una mayor prioridad que a las variables Z en el *branching* mejora el desempeño del algoritmo ya que disminuye la cantidad de nodos que se generan en el árbol de búsqueda haciéndolo también más balanceado.

4.3.1. Comparación de desempeño

En las tablas (4.5) y (4.6) se muestran las comparaciones entre los resultados obtenidos al resolver las mismas instancias sin incorporar las variables X ni V (SXV), incorporando solo las variables X (X), incorporando solo las variables V (V) e incorporando ambas variables

(XV).

Las ejecuciones de la variante XV se hicieron utilizando la siguiente prioridad para el *branching*:

$$prioridad(V) > prioridad(X) > prioridad(Z)$$

Esta decisión fue tomada basándose en que el desempeño de la variante V es mucho mejor que la variante X , como podrá verse en las siguientes dos tablas. Todas las variantes se han ejecutado utilizando **FB1**, **OST**, **HI** y con un límite de tiempo de 5 horas.

Para calcular los porcentajes de variación del valor objetivo y de los tiempos de ejecución de cada instancia utilizamos las siguientes fórmulas:

$$\% \text{ Variación Valor Objetivo } X = (X - SXV)/SXV$$

$$\% \text{ Variación Valor Objetivo } V = (V - SXV)/SXV$$

$$\% \text{ Variación Valor Objetivo } XV = (XV - SXV)/SXV$$

$$\% \text{ Variación Tiempo } X = (X - SXV)/SXV$$

$$\% \text{ Variación Tiempo } V = (V - SXV)/SXV$$

$$\% \text{ Variación Tiempo } XV = (XV - SXV)/SXV$$

El porcentaje que se muestra en la tabla es el promedio de las variaciones para cada caso. Los valores negativos representan mejoras sobre la versión SXV.

Caso	V	C	#Inst.	% Variación Valor Objetivo		
				X	V	XV
v3_c12	3	12	6	1.434	0.000	0.000
v3_c14	3	14	6	4.313	-0.063	-0.063
v3_c17	3	17	6	-2.853	-5.481	-5.481
v4_c18	4	18	6	10.195	-2.036	8.047
v5_c21	5	21	6	0.760	0.036	0.760

Tabla 4.5: Variación valor objetivo con variables X y V

Caso	V	C	#Inst.	% Variación Tiempo		
				X	V	XV
v3_c12	3	12	6	31.632	-99.199	-98.788
v3_c14	3	14	6	0.000	-94.590	-92.918
v3_c17	3	17	6	0.000	-85.078	-82.433

Tabla 4.6: Variación tiempo con variables X y V

Todas las instancias de los casos v4_c18 y v5_c21 terminaron por timeout de 5 horas (incluso con las ejecuciones con la variante SXV).

Analizando ambas tablas podemos observar que la variante X empeora la performance del algoritmo, empeorando los valores objetivos en la mayoría de los casos y los tiempos de las instancias del caso más chico.

En la tabla (4.6) puede verse que las variantes V y XV mejoran drásticamente los tiempos de ejecución de todas las instancias que terminan antes que se cumpla el tiempo límite impuesto de 5 horas. Pero de la tabla (4.5) se desprende que la variante V mejora los valores objetivos en la mayoría de los casos lo cual no ocurre con la variante XV. El promedio de las variaciones de valores objetivos para V es de -1.509 mientras que el de XV es de +0.652.

De lo anteriormente detallado se desprende que la mejor opción a utilizar de aquí en adelante entre las variantes SXV, X, V y XV es V.

4.4. Heurística Primal

Cuando la solución de la relajación lineal es buena, explotar esta información puede generar soluciones factibles de muy buena calidad. Mediante procedimientos heurísticos se construyen soluciones enteras basadas en el óptimo fraccionario de la relajación. Estos procedimientos cuentan con más información del problema que al comienzo del algoritmo y esto hace suponer que serán capaces de encontrar mejores soluciones.

Las heurísticas que hacen uso de la solución fraccionaria para crear una solución factible son conocidas como heurísticas primales. El ejemplo más simple de una heurística primal es redondear las variables fraccionarias a valores enteros factibles. Si bien las heurísticas primales obtienen generalmente buenas cotas superiores, es necesario encontrar un balance correcto entre la efectividad del procedimiento y el tiempo de cómputo requerido.

En nuestro algoritmo, experimentamos tres criterios para construir una solución entera

a partir de la solución de las relajaciones. Todos pertenecen a la clase de algoritmos denominados golosos o voraces, los cuales para resolver un determinado problema, eligen la opción óptima local en cada paso con la esperanza de llegar a una solución general óptima. Este esquema algorítmico brinda muchas veces soluciones aceptablemente buenas siendo el que menos dificultades plantea a la hora de implementar y comprobar su funcionamiento. Normalmente se aplica a los **problemas de optimización**.

4.4.1. Heurística Primal en Serie

La idea básica de esta heurística es construir los recorridos completos para cada visitador en forma serial, es decir, que se calcula el recorrido de un visitador y una vez finalizado se comienza a calcular el recorrido del siguiente visitador.

Dada una solución fraccionaria seleccionamos la variable $Z_{v,0,c}$ con mayor valor para todo v y todo c . Dicha variable determina el primer visitador al cual se le armará el recorrido y el primer comercio de su recorrido. Esto se fundamenta en que esta variable es un buen candidato a valer 1 en la solución entera (recordar que las variables Z pueden tomar valores entre 0 y 1 en la relajación lineal).

A partir del primer comercio asignado se busca el próximo comercio para asignar al recorrido minimizando la siguiente fórmula:

$$(1 - Z_{v,c,x}) * d_{c,x}$$

donde x son todos los comercios no asignados a un recorrido. Esta minimización determina que x será el próximo comercio a asignar en el recorrido del visitador. Dicha minimización tiene un doble objetivo que es maximizar el valor de $Z_{v,c,x}$ (por la razón anteriormente mencionada) y a su vez minimizar el valor de $d_{c,x}$ que es la distancia que existe entre los comercios c y x . Esto último se realiza para obtener la menor distancia local (recordar que estamos utilizando un algoritmo goloso).

Con el mismo procedimiento se determinan los restantes comercios del recorrido del visitador. A partir del último comercio asignado al recorrido, se determina el primer comercio del próximo visitador utilizando el mismo procedimiento de minimización antes mencionado y así sucesivamente hasta completar los recorridos de todos los visitadores.

Pseudocódigo de la Heurística Primal en Serie

Definición de variables:

SEVO es el valor objetivo de la mejor solución entera obtenida hasta el momento.

SF es la solución fraccionaria que recibe la heurística.

V es la cantidad de visitantes.

C es la cantidad de comercios.

CI es el comercio inicial.

CA es el comercio actual.

VA es el visitador actual.

PC es el próximo comercio.

Ver pseudocódigo en hoja 34

4.4.2. Heurística Primal en Paralelo 1

La idea básica de esta heurística es construir los recorridos de todos los visitantes en forma paralela. Para esto asignamos el primer comercio del recorrido de cada uno de los visitantes, luego el segundo comercio y así sucesivamente hasta completar todos los recorridos.

Dada una solución fraccionaria seleccionamos la variable $Z_{v,0,c}$ con mayor valor para todo v y todo c . Dicha variable determina que c será el primer comercio del recorrido del visitador v . Para determinar el primer comercio del recorrido del resto de los visitantes, se realiza el mismo proceso teniendo en cuenta que v no sea ninguno de los visitantes a los cuales ya se les haya asignado su primer comercio y que c no sea alguno de los comercios ya asignados.

A partir del primer comercio asignado a cada recorrido se busca el segundo comercio de un visitador minimizando la siguiente fórmula:

$$(1 - Z_{v,c,x}) * d_{c,x}$$

donde v es alguno de los visitantes a los cuales aún no se asignó el segundo comercio, c es el primer comercio asignado al visitador v elegido y x son todos los comercios no asignados a ningún recorrido. Esta minimización determina que x será el segundo comercio a asignar en el recorrido del visitador v . Tal como ocurre con la heurística primal en serie, esta minimización tiene el doble objetivo de maximizar el valor de $Z_{v,c,x}$ y a su vez minimizar el valor de $d_{c,x}$. Con el mismo procedimiento se asignan los segundos comercios de los visitantes restantes.

Utilizando la misma lógica se determinan los restantes comercios del recorrido de cada visitador.

Heurística Primal en Serie

```

inicializar la variable SEVO en infinito
hacer C x V veces (*)
  inicializar todas las variables Z de la solución entera a construir con valor 0
  marcar todos los comercios como sin asignar
  marcar todos los visitantes como con recorrido pendiente de definir
  hacer V veces
    buscar el mayor  $Z_{(v,0,c)}$  para todo comercio  $c$  que no haya sido asignado aún y para todo visitante  $v$  que
    tenga pendiente de definir su recorrido
    verificar no comenzar el recorrido del primer visitante con el mismo comercio que empezó en otra
    iteración anterior de (*)
    asignar a VA el valor de  $v$ 
    asignar a CI el valor de  $c$ 
    asignar la variable  $Z_{(VA,0,CI)}$  con el valor 1
    marcar CI como comercio asignado
    asignar a CA el valor de CI
    hacer  $(C/V - 1)$  veces
      buscar el comercio  $c$  que minimiza distancia  $(CA,c) * (1 - Z_{(VA,CA,c)})$  entre los comercios que aún no han
      sido asignados
      asignar a PC el valor de  $c$ 
      asignar la variable  $Z_{(VA,CA,PC)}$  con el valor 1
      marcar PC como comercio asignado
      asignar a CA el valor de PC
      asignar la variable  $Z_{(VA,CA,C+1)}$  con el valor 1
      marcar el visitante VA como que ya tiene el recorrido definido
      asignar los valores a las variables T a partir de los asignados a las variables Z
      calcular el valor de la función objetivo en la solución entera obtenida
      comparar este valor objetivo con SEVO y actualizar el valor de SEVO
      asignar los valores a las variables X y V a partir de los asignados a las variables Z

```

Cabe recordar que puede darse el caso que no todos los visitantes tengan la misma cantidad de comercios en el recorrido (con una diferencia de a lo sumo un comercio). Para asignar estos comercios restantes se utiliza el mismo procedimiento. De esta manera los comercios restantes serán asignados a los C módulo V visitantes que minimicen la fórmula.

Pseudocódigo de la Heurística Primal en Paralelo 1

$SEVO$ es el valor objetivo de la mejor solución entera obtenida hasta el momento.

SF es la solución fraccionaria que recibe la heurística.

V es la cantidad de visitantes.

C es la cantidad de comercios.

Ver pseudocódigo en hoja 36

4.4.3. Heurística Primal en Paralelo 2

La idea básica de esta heurística es construir los recorridos de todos los visitantes en forma paralela. Para esto asignamos el primer comercio del recorrido de cada uno de los visitantes, luego el segundo comercio y así sucesivamente hasta completar todos los recorridos.

El primer comercio del recorrido de cada visitante es seleccionado al azar.

A partir del primer comercio asignado a cada recorrido se busca el segundo comercio de cada visitante minimizando la siguiente fórmula:

$$(1 - X_{c,x}) * d_{c,x}$$

donde c es el primer comercio asignado al visitante actual (los visitantes están ordenados ascendentemente) y x son todos los comercios no asignados a ningún recorrido. Esta minimización determina que x será el segundo comercio a asignar en el recorrido del visitante actual. Tal como ocurre con la heurística primal paralelo 1, esta minimización tiene un doble objetivo pero en este caso se trata de maximizar el valor de $X_{c,x}$ y a su vez minimizar el valor de $d_{c,x}$.

Con el mismo procedimiento se determinan los restantes comercios del recorrido de cada visitante.

Heurística Primal en Paralelo 1

inicializar la variable *SEVO* en infinito
 hacer $C \times V$ veces (*)
 inicializar todas las variables Z de la solución entera a construir con valor 0
 marcar todos los comercios como sin asignar
 marcar todos los visitantes como con recorrido pendiente de definir
 hacer V veces
 buscar el mayor $Z_{v,0,c}$ para todo comercio c que no haya sido asignado aún y para todo visitante v que tenga pendiente de definir su primero comercio del recorrido
 verificar no comenzar el recorrido del primer visitante con el mismo comercio que empezó en otra iteración anterior de (*)
 asignar la variable $Z_{v,0,c}$ con el valor 1
 marcar c como comercio asignado
 marcar v como visitante ya inicializado
 asignar c como primer comercio de v
 hacer $(C/V - 1)$ veces para definir los restantes j comercios (con $j = C/V - 1$)
 marcar todos los visitantes como con comercio j -ésimo pendiente de definir
 hacer V veces
 buscar el comercio c y el visitante v que minimiza distancia(*ultimo-comercio-asignado-recorrido-v,c*)*
 (1 - $Z_{v,ultimo-comercio-asignado-recorrido-v,c}$) entre los comercios que aún no han sido asignados y los v que aún no tienen asignado el j -ésimo comercio
 asignar la variable $Z_{v,ultimo-comercio-asignado-recorrido-v,c}$ con el valor 1
 asignar c como ultimo comercio asignado a v
 marcar c como comercio asignado
 marcar v como visitante con j -ésimo comercio ya asignado
 hacer ($C \text{ mod } V$) veces (equivalente a los j comercios no asignados)
 buscar el comercio c y el visitante v que minimiza distancia(*ultimo-comercio-asignado-recorrido-v,c*)*
 (1 - $Z_{v,ultimo-comercio-asignado-recorrido-v,c}$) entre los comercios que aún no han sido asignados y los v que aún no tienen asignado el j -ésimo comercio
 asignar la variable $Z_{v,ultimo-comercio-asignado-recorrido-v,c}$ con el valor 1
 asignar c como ultimo comercio asignado a v
 marcar c como comercio asignado
 marcar v como visitante con j -ésimo comercio ya asignado
 asignar los valores a las variables T a partir de los asignados a las variables Z
 calcular el valor de la función objetivo en la solución entera obtenida
 comparar este valor objetivo con *SEVO* y actualizar el valor de *SEVO*
 asignar los valores a las variables X y V a partir de los asignados a las variables Z

Pseudocódigo de la Heurística Primal en Paralelo 2

$SEVO$ es el valor objetivo de la mejor solución entera obtenida hasta el momento.

SF es la solución fraccionaria que recibe la heurística.

V es la cantidad de visitantes.

C es la cantidad de comercios.

Ver pseudocódigo en hoja 38

Comparación de desempeño

A continuación se muestra la comparación entre los resultados obtenidos al resolver las mismas instancias sin utilizar la heurística primal (SHP), utilizando la heurística primal en serie (HPS) y utilizando ambas variantes de heurística primal en paralelo (HPP1 y HPP2). Todas las variantes se han ejecutado utilizando **FB1**, **OST**, **HI**, **V** y con un límite de tiempo de 5 horas.

Para calcular los porcentajes de variación del valor objetivo y de los tiempos de ejecución de cada instancia utilizamos las siguientes fórmulas:

$$\% \text{ Variación Valor Objetivo } HPS = (HPS - SHP) / SHP$$

$$\% \text{ Variación Valor Objetivo } HPP1 = (HPP1 - SHP) / SHP$$

$$\% \text{ Variación Valor Objetivo } HPP2 = (HPP2 - SHP) / SHP$$

$$\% \text{ Variación Tiempo } HPS = (HPS - SHP) / SHP$$

$$\% \text{ Variación Tiempo } HPP1 = (HPP1 - SHP) / SHP$$

$$\% \text{ Variación Tiempo } HPP2 = (HPP2 - SHP) / SHP$$

El porcentaje que se muestra en la tabla es el promedio de las variaciones para cada caso. Los valores negativos representan mejoras sobre la versión SHP.

Del análisis de la tabla (4.7) puede verse que las 3 variantes mejoran sustancialmente los valores objetivos en los casos más grandes. Como estas mejoras son muy similares entre sí, no podemos decidir cual de las 3 variantes es la mejor.

Luego analizando la tabla (4.8) vemos que la variante HPS mejora los tiempos en todas los casos a excepción de los de menor tamaño. Estos últimos son justamente los menos

Heurística Primal en Paralelo 2

```

inicializar la variable SEVO en infinito
hacer C x V veces
  inicializar todas las variables Z de la solución entera a construir con valor 0
  marcar todos los comercios como sin asignar
  asignar al azar el primer comercio al recorrido de cada visitador
  hacer (C / V - 1) veces para definir los restantes j comercios (con j = C/V - 1)
  marcar todos los visitadores como con comercio j-ésimo pendiente de definir
  hacer V veces
    buscar el comercio c y el visitador v que minimiza distancia(ultimo_comercio_asignado_recorrido_v, c)*
    (1 - X_ultimo_comercio_asignado_recorrido_v, c) entre los comercios que aún no han sido asignados y los v que aún
    no tienen asignado el j-ésimo comercio
    asignar la variable Z_{v,ultimo_comercio_asignado_recorrido_v,c} con el valor 1
    asignar c como ultimo comercio asignado a v
    marcar c como comercio asignado
    marcar v como visitador con j-ésimo comercio ya asignado
    hacer (C mod V) veces (equivalente a los j comercios no asignados)
    buscar el comercio c y el visitador v que minimiza distancia(ultimo_comercio_asignado_recorrido_v, c)*
    (1 - X_ultimo_comercio_asignado_recorrido_v, c) entre los comercios que aún no han sido asignados y los v que aún no
    tienen asignado el j-ésimo comercio
    asignar la variable Z_{v,ultimo_comercio - asignado - recorrido - v, c} con el valor 1
    asignar c como ultimo comercio asignado a v
    marcar c como comercio asignado
    marcar v como visitador con j-ésimo comercio ya asignado
    asignar los valores a las variables T a partir de los asignados a las variables Z
    calcular el valor de la función objetivo en la solución entera obtenida
    comparar este valor objetivo con SEVO y actualizar el valor de SEVO
    asignar los valores a las variables X y V a partir de los asignados a las variables Z

```

Caso	V	C	#Inst.	% Variación Valor Objetivo		
				HPS	HPP1	HPP2
v3_c12	3	12	6	0.000	0.000	0.000
v3_c14	3	14	6	0.000	0.000	0.000
v3_c17	3	17	6	0.000	0.000	0.000
v4_c18	4	18	6	-21.900	-22.024	-22.395
v5_c21	5	21	6	-24.732	-24.386	-24.290

Tabla 4.7: Variación valor objetivo con Heurísticas Primales

Caso	V	C	#Inst.	% Variación Tiempo		
				HPS	HPP1	HPP2
v3_c12	3	12	6	35.963	43.553	107.997
v3_c14	3	14	6	-10.194	6.993	46.995
v3_c17	3	17	6	-0.585	15.686	40.564
v4_c18	4	18	6	-32.143	-20.431	-35.713
v5_c21	5	21	6	-4.721	-1.265	-2.538

Tabla 4.8: Variación tiempo con Heurísticas Primales

significativos para el análisis ya que los algoritmos implementados apuntan a resolver instancias grandes. También puede verse que las mejoras provistas por HPS son significativamente mejores que las de HPP1 y HPP2.

De lo anteriormente detallado se desprende que la mejor opción a utilizar de aquí en adelante entre las variantes SHP, HPS, HPP1 y HPP2 es HPS.

4.5. Rompimiento de Simetría

Las soluciones simétricas son distintas soluciones factibles del modelo de programación lineal entera que representan la misma solución del problema real. Evitar la existencia de este tipo de soluciones modificando el modelo puede ayudar a veces a que un algoritmo revise menos soluciones factibles. La eliminación de estas soluciones implica agregar restricciones al modelo, con lo cual a veces produce un efecto contraproducente dado que la dificultad de resolver el nuevo modelo no se justifica frente a los perjuicios de tratar con soluciones simétricas.

Profundizando en la característica del modelo sobre la **indistinguibilidad de los visitantes** mencionada anteriormente, detectamos que podía realizarse una nueva optimización

con el objetivo de reducir el espacio de soluciones del problema. Estudiamos 2 variantes para lograr esta optimización que si bien no rompen la simetría de manera completa, achican en buena medida el espacio de soluciones eliminando soluciones simétricas.

La primera consiste en agregar restricciones para establecer un orden en las distancias recorridas por los visitantes. La idea es que mediante estas restricciones se cumplan las siguientes desigualdades:

$$D_{v_1} \leq D_{v_2} \leq \dots \leq D_{v_n}$$

donde n es la cantidad de visitantes.

En total se agregan $n - 1$ restricciones $D_{v_i} \leq D_{v_{i+1}}$ para $i = 1, \dots, n - 1$.

Suponiendo que todas las distancias de los recorridos de todos los visitantes sean distintas ($D_{v_1} < D_{v_2} < \dots < D_{v_n}$), puede verse que estas restricciones nos permiten reducir enormemente el espacio de soluciones ya que por cada solución posible del problema incluyendo las restricciones de rompimiento de simetría existen $n!$ soluciones del problema sin utilizar dichas restricciones. Por ejemplo, en el caso de tener tan solo 5 visitantes la reducción es de 120 a 1.

Si bien las restricciones que se agregan son las originalmente mencionadas (con \leq), lo que es interesante notar acá es que en la práctica es muy poco probable que la distancia del recorrido de dos visitantes sean iguales.

Llamaremos a esta versión **RDS1** (Rompimiento de Simetría 1).

La segunda variante consiste en agregar restricciones para establecer un orden entre el primer comercio del recorrido de cada visitante utilizando su número. La idea es que mediante estas restricciones se cumplan las siguientes desigualdades:

$$C_{v_1} \leq C_{v_2} \leq \dots \leq C_{v_n}$$

donde n es la cantidad de visitantes y C_{v_x} es el primer comercio en el recorrido del visitante x .

En total se agregan $n-1$ restricciones $C_{v_i} \leq C_{v_{i+1}}$ para $i = 1, \dots, n - 1$.

Puede verse que estas restricciones también nos permiten reducir enormemente el espacio de soluciones ya que por cada solución posible del problema incluyendo las restricciones de rompimiento de simetría existen $n!$ soluciones del problema sin utilizar dichas restricciones.

Llamaremos a esta versión **RDS2** (Rompimiento de Simetría 2).

4.5.1. Comparación de desempeño

A continuación se muestra la comparación entre los resultados obtenidos al resolver las mismas instancias sin utilizar rompimiento de simetría (SRDS), utilizando RDS1 y RDS2. Todas las variantes se han ejecutado utilizando **FB1**, **OST**, **HI**, **V**, **HPS** y con un límite de tiempo de 5 horas.

Para calcular los porcentajes de variación del valor objetivo y de los tiempos de ejecución de cada instancia utilizamos las siguientes fórmulas:

$$\% \text{ Variación Valor Objetivo } RDS1 = (RDS1 - SRDS) / SRDS$$

$$\% \text{ Variación Valor Objetivo } RDS2 = (RDS2 - SRDS) / SRDS$$

$$\% \text{ Variación Tiempo } RDS1 = (RDS1 - SRDS) / SRDS$$

$$\% \text{ Variación Tiempo } RDS2 = (RDS2 - SRDS) / SRDS$$

El porcentaje que se muestra en la tabla es el promedio de las variaciones para cada caso.

Caso	V	C	#Inst.	% Variación Valor Objetivo	
				RDS1	RDS2
v3_c14	3	14	6	0.000	0.000
v3_c17	3	17	6	0.000	0.000
v4_c18	4	18	6	0.068	0.449
v5_c21	5	21	6	0.294	0.005

Tabla 4.9: Variación valor objetivo con Rompimiento de Simetría

Caso	V	C	#Inst.	% Variación Tiempo	
				RDS1	RDS2
v3_c14	3	14	6	59.353	5.776
v3_c17	3	17	6	77.053	24.157
v4_c18	4	18	6	52.239	58.392
v5_c21	5	21	6	5.820	5.820

Tabla 4.10: Variación tiempo con Rompimiento de Simetría

Claramente del análisis de ambas tablas puede verse que las variantes no ofrecen ninguna mejora con respecto a SRDS sino que por el contrario empeoran los resultados obtenidos anteriormente, por lo que no se utilizará ninguna de aquí en adelante.

4.6. Desigualdades válidas

El objetivo de aplicar planos de corte es obtener relajaciones lineales más ajustadas, de manera de poder conseguir mejores cotas duales. Al mismo tiempo, se debe considerar que, la complejidad del algoritmo para resolver el problema de separación debe justificarse de acuerdo al beneficio obtenido gracias a las desigualdades agregadas. Además se deben tener en cuenta la cantidad de desigualdades generadas, ya que agregando una gran cantidad de desigualdades, el tiempo de resolución de las relajaciones se incrementa así como también el espacio utilizado en memoria.

En esta sección se presentan familias de desigualdades válidas que serán utilizadas como planos de corte.

El problema de ruteo de vehículos capacitado (PRVC) fue extensamente estudiado desde el punto de vista poliedral por diversos autores [1, 2, 7] y hay caracterizadas varias familias de desigualdades válidas. Dada la relación entre ese problema y el abordado en este trabajo, las desigualdades válidas para el PRVC pueden ser fácilmente adaptadas al PRV.

4.6.1. Desigualdad de Capacidad

Dado un conjunto de clientes $S \subseteq \{1, \dots, C\}$, si definimos $Q = \lceil \frac{C}{V} \rceil$ la desigualdad de Capacidad para nuestro problema se puede expresar como

$$\sum_{\substack{c_1 \in S \\ c_2 \in S}} \sum_{v \in V} Z_{v,c_1,c_2} \leq |S| - \left\lceil \frac{|S|}{Q} \right\rceil \quad S \subseteq \{1, \dots, C\} \quad (4.1)$$

Claramente estas desigualdades dominan a las EST.

4.6.2. Desigualdades *Star*

Las desigualdades *Star* fueron propuestas en [1] para el PRVC con demanda unitaria. La estructura de estas desigualdades es

$$B \sum_{\substack{c_1 \in N \\ c_2 \notin N}} \sum_{v \in \{1, \dots, V\}} Z_{v,c_1,c_2} - A \sum_{\substack{c_1 \in O \\ c_2 \in T}} \sum_{v \in \{1, \dots, V\}} Z_{v,c_1,c_2} \leq L$$

donde $N \subset \{1, \dots, C\}$ es el *núcleo*, $T \subset \{1, \dots, C\} \setminus N$ son los *satélites*, $O \subset \{1, \dots, C\} \setminus N$ los *conectores* y A, B y L son constantes que depende de $|N|$, $|T|$ y $|O|$.

4.6.3. Desigualdades *Comb*

Las desigualdades *Comb* son una familia muy conocida de desigualdades válidas para el problema del viajante de comercio introducidas por Grötschel y Padberg [6]. Estas han probado ser muy útiles a la hora de utilizarse como planos de cortes. Varios trabajos en la literatura las adaptan para el problema de ruteo de vehículos capacitado [1, 2]. Nosotros trabajamos con el enfoque dado por Araque [1].

Para cualquier conjunto $S \subset \{0, 1, \dots, C + 1\}$, sea

$$\hat{k}(S) = \begin{cases} \left\lceil \frac{|S|}{Q} \right\rceil & \text{si el depósito está en } S \\ \left\lfloor \frac{C-|S|}{Q} \right\rfloor & \text{en caso contrario} \end{cases}$$

Adicionalmente, sea

$$S(H, T_1, \dots, T_t) = \sum_{j=1}^t (\hat{k}(T_j \cap H) + \hat{k}(T_j \setminus H) + \hat{k}(T_j)).$$

Una desigualdad *comb* es un conjunto de vértices $H \subset \{1, \dots, C\}$, llamado el **mango**, y otros conjuntos de vértices $T_1, \dots, T_t \subset \{0, \dots, C + 1\}$ con $t \geq 2$ llamados **dientes**, tales que:

- $H \cap T_j \neq \emptyset$ y $T_j \setminus H \neq \emptyset$, para $j = 1, \dots, t$.
- Para cada par i, j con $i, j \in \{1, \dots, t\}$, $i \neq j$, $T_i \cap T_j \subset H$ o $T_i \cap T_j \cap H = \emptyset$.

Si $S(H, T_1, \dots, T_t)$ es impar, la siguiente desigualdad es válida

$$\sum_{\substack{c_1 \in H \\ c_2 \notin H}} \sum_{v \in \{1, \dots, V\}} Z_{v, c_1, c_2} + \sum_{l=1}^t \sum_{\substack{i \in T_l \\ j \notin T_l}} \sum_{v \in \{1, \dots, V\}} Z_{v, c_1, c_2} \geq S(H, T_1, \dots, T_t) + 1$$

Esta es una versión más general que la estudiada para el TSP, porque permite la intersección entre los dientes y no requiere que el número de dientes sea impar.

4.6.4. Algoritmos de Separación

La aplicación de un algoritmo de planos de corte tiene una etapa decisiva, la etapa de *separación*. Debemos ser capaces de encontrar desigualdades violadas, es decir, identificar una desigualdad que *corte* la solución fraccionaria actual, o probar que tal desigualdad no existe.

Para lograr una buena performance global los procedimientos de separación deben ser rápidos. En algunos casos este objetivo puede cumplirse con algoritmos exactos, pero en

otros es necesario implementar heurísticas. Estas últimas no aseguran que siempre se pueda detectar desigualdades violadas, pero es una solución de compromiso ante el alto costo computacional de algunos problemas de separación.

En nuestra implementación desarrollamos procedimientos de separación para las desigualdades válidas mencionadas en la sección anterior. Para las desigualdades Capacidad, *Star* y *Comb* utilizamos el paquete de algoritmos de separación desarrollado por Lechtford, Eglese y Lysgaard [7, 8] disponible en <http://www.hha.dk/~lys/>.

4.6.5. Evaluación de las Familias de Desigualdades Válidas

Mediante la experimentación computacional tratamos de obtener conclusiones sobre el comportamiento de cada una de las familias considerando diferentes criterios y alternativas.

En los experimentos computacionales que hicimos hemos considerado las desigualdades EST del modelo original como planos de corte. Se probaron las familias de desigualdades válidas utilizando un algoritmo de planos de corte. El algoritmo termina cuando no es posible encontrar desigualdades violadas o cuando alcanza un máximo de 50 iteraciones. Una desigualdad es incorporada como un plano de corte cuando la violación es mayor a 0,01. Las instancias de prueba son las descriptas al inicio del capítulo 4.

Para el análisis de las familias utilizamos todas las combinaciones posibles entre ellas y además una adicional llamada “Comb si no Capacidad ni Star”. Esta última busca en cada iteración cortes para las familias Capacidad y Star y solamente en el caso de no haber encontrado ningún corte busca cortes Comb.

En las tablas que se verán en las siguientes secciones utilizaremos las siguiente abreviaturas para cada combinación de familias de desigualdades válidas:

- CB: solo Comb
- CP: solo Capacidad
- ST: solo Star
- CB+CP: Comb y Capacidad
- CB+ST: Comb y Star
- CP+ST: Capacidad y Star
- CB+CP+ST: Comb, Capacidad y Star
- CB/CP+ST: Comb si no Capacidad ni Star

Tiempo de Separación

Si los algoritmos de separación insumen mucho tiempo en relación al beneficio obtenido en el incremento de la cota inferior, no vale la pena incluirlos en el algoritmo. Este no es un factor que influye en nuestra implementación. Ejecutando instancias con cantidad de clientes entre 14 y 21, el total de tiempo invertido en estos procedimientos para la mejor combinación de familias fue del 20 % en el peor caso y del 4 % en promedio.

Cantidad de Cortes por Familia

El total de cortes violados encontrados por cada algoritmo de separación es una medida a tener en cuenta. Si la cantidad de cortes violados es pequeña puede ser porque el algoritmo de separación no sea eficiente o porque la familia no sea violada por las soluciones fraccionarias. En el primer caso, debería implementarse un algoritmo mejor. En el segundo caso, salvo que la adición de un número pequeño de desigualdades incremente significativamente la función objetivo, indicaría que la familia no aporta a la performance del algoritmo. Cabe señalar que un número grande de desigualdades válidas violadas no garantiza que su inclusión favorezca el incremento de la función objetivo, además de agrandar el tamaño de las relajaciones y por lo tanto requerir más tiempo de CPU.

Para evaluar esta situación ejecutamos un máximo de 100 cortes de la familia Star y $C \times V \times 5$ cortes de las familias Comb y Capacidad por iteración. Estos valores surgieron de distintas pruebas realizadas agregando distintas cantidades de restricciones incluso sin poner un tope para las mismas. En la tabla (4.11) mostramos los resultados obtenidos.

Caso	CB	CP	ST	CB+CP	CB+ST	CP+ST	CB+CP+ST	CB/CP+ST
v3_c14	0	18	0	18	0	19	19	19
v3_c17	0	24	0	26	0	36	38	37
v4_c18	0	22	0	24	0	36	40	36
v5_c21	0	21	0	22	0	35	36	35

Tabla 4.11: Cantidad de Cortes por Combinación de Familias

En la tabla puede verse que la familia *Capacidad* es la que más cortes encuentra, ya sea que se la utilice en forma independiente o en combinación con otras familias.

Evolución de la Cota Inferior y Tiempo Total

Para la evaluación final de la eficiencia de las distintas familias de desigualdades válidas utilizamos como criterio la evolución de la cota inferior de la relajación lineal cuando es ajustada por la adición de éstas. A mayor incremento, mejor la calidad de la desigualdad

válida. Como ya mencionamos antes, hay que buscar un balance entre diferentes aspectos. No siempre vale la pena insumir mucho tiempo con el fin de obtener calidad de cota.

El análisis lo hemos hecho por familia. Como disponemos de 3 familias de cortes, cabe preguntarse si existe alguna de ellas que tenga una mejor performance comparada con las otras. De todos los factores que estudiamos, este fue el de mayor dificultad y que nos llevó el mayor tiempo de experimentación para obtener una respuesta. Experimentamos con muchas instancias de diferente cantidad de clientes, buscando alguna característica que nos indicara la ventaja de un corte sobre otro.

Caso	S/Cortes	CB		CP		ST		CB+CP	
	Tiempo	Tiempo	%Gap	Tiempo	%Gap	Tiempo	%Gap	Tiempo	%Gap
v3_c14	7.17	5.70	0.00	1.59	51.77	5.69	0.00	1.94	51.77
v3_c17	36.89	36.97	0.00	17.87	45.00	36.89	0.00	6.77	45.02
v4_c18	9.25	9.32	0.00	5.45	39.16	9.29	0.00	12.87	39.16
v5_c21	14.98	15.04	0.00	11.75	28.04	15.03	0.00	14.33	28.04

Caso	CB+ST		CP+ST		CB+CP+ST		CB/CP+ST	
	Tiempo	%Gap	Tiempo	%Gap	Tiempo	%Gap	Tiempo	%Gap
v3_c14	5.72	0.00	2.03	51.77	1.63	51.77	2.05	51.77
v3_c17	36.99	0.00	4.40	45.00	6.52	45.02	4.24	45.02
v4_c18	9.35	0.00	4.20	39.16	4.11	39.16	4.24	39.16
v5_c21	15.07	0.00	9.05	28.04	13.34	28.04	9.13	28.04

Tabla 4.12: Evolución Cota Inferior y Tiempo Total

En la tabla 4.12 reportamos nuestra experiencia con cada familia de cortes sobre instancias entre 14 y 21 clientes. Para cada tamaño consideramos el promedio sobre 5 instancias. Indicamos el porcentaje de *gap* entre el valor de la relajación inicial y la cota inferior alcanzada utilizando esa familia como cortes y el tiempo total.

La reducción mínima de *gap* es del orden del 0% y la máxima del 52%. Se ve que no todas las combinaciones de familias influyen en el mejoramiento de la cota inferior.

En todos los casos las mejores cotas fueron obtenidas cuando se utiliza la combinación *Comb si no Capacidad ni Star*, obteniendo en promedio un *gap* de 41% en 4.91 segundos. La combinación *Comb, Capacidad y Star* también presentó una alta eficiencia, logró el mismo *gap* de 41% pero en 6.40 segundos para las mismas instancias.

Son varios los criterios que utilizamos para evaluar las desigualdades válidas: evolución de la cota inferior, tiempos de los algoritmos de separación y violación de los cortes. Todos

ellos están relacionados y nos inducen a conclusiones que no entran en conflicto. Debido al pequeño *gap* entre el valor óptimo y la cota inferior obtenida por el procedimiento de planos de corte, estos experimentos confirman que las familias de desigualdades válidas utilizadas son fuertes y que los algoritmos de separación son eficientes ya que permiten generar buenas cotas inferiores con tiempos de ejecución razonables.

4.7. Comparación CPLEX vs BC-V

A continuación se muestra la comparación entre los resultados obtenidos al resolver las mismas instancias utilizando el algoritmo CPLEX y el algoritmo BC-V usando la mejor combinación de familias de cortes (Comb si no Capacidad ni Star):

Caso	% Gap Objetivo vs. Cota inferior		% Gap entre Objetivos (1)	% Gap entre cotas Inferiores (2)	% Gap entre Tiempos (1)	% Comparación Nodos recorridos
	CPLEX	BC-V				
v3_c14	13.4596 {2}	6.5410 {3}	0.0000	3.8877	-51.5662	0.3153
v3_c17	17.4996	2.0869 {2}	-0.9252	13.6967	-33.4181	0.3736
v4_c18	50.6900	21.3162	-2.2706	15.0851	(3)	0.3647
v5_c21	90.2940	43.6443	13.6378	21.1661	(3)	0.1654
Promedio	42.9858	18.3971	2.6105	13.4589	-21.2461	0.3047

(1) Los valores negativos representan mejoras de BC-V sobre CPLEX

(2) Los valores positivos representan mejoras de BC-V sobre CPLEX

(3) Todos los casos terminaron por timeout tanto en BC-V como en CPLEX

Entre llaves “{ }” se indica la cantidad de instancias que terminaron sin llegar al límite máximo de tiempo

Tabla 4.13: CPLEX vs. BC-V

En la segunda y tercer columna puede verse que el promedio de gap obtenido entre el valor objetivo y la cota inferior es mucho mejor en BC-V que en CPLEX.

En la cuarta y sexta columna puede verse que en los primeros 3 casos mejoran mucho los tiempos y en menor medida los valores objetivos obtenidos por el BC-V. El último caso empeora el valor objetivo pero las cotas inferiores obtenidas son mucho mejores.

En la última columna puede verse que la cantidad de nodos recorridos por BC-V disminuyó drásticamente con respecto a CPLEX. En promedio BC-V recorrió un nodo cada 297 que recorrió CPLEX.

4.8. Conclusiones y trabajo futuro

En esta tesis desarrollamos un algoritmo para la resolución del *Problema de los Visitadores* utilizando modelos de programación lineal entera mixta. Este es un nuevo problema de optimización combinatoria que surgió del análisis de un problema real.

Con el objetivo de desarrollar un algoritmo *Branch-and-Cut*, inicialmente propusimos una formulación de programación lineal entera que modelaba el problema estudiado. Para mejorar los tiempos de resolución que se obtienen con CPLEX para este modelo, se estudiaron distintos factores que hacen a los algoritmos *Branch-and-Bound* y *planos de corte*.

Como primera medida desarrollamos una *heurística inicial* con el fin de obtener una cota inicial suficientemente buena como para permitir achicar los tiempos de ejecución. Esta cota nos permitió realizar podas sobre el árbol de búsqueda mejorando en la práctica el tiempo de resolución de las instancias.

Luego nos focalizamos en el *branching*, para lo cual introdujimos las variables X y V. De la práctica se desprendió que la mejor opción era incluir solamente las variables V asignándole la mayor prioridad en el branching.

Adicionalmente, desarrollamos 3 variantes de *heurística primal*. Tras el análisis de dichas variantes los mejores resultados fueron obtenidos con la *Heurística Primal en Serie*.

Luego se analizó el problema de simetría en las soluciones para lo cual desarrollamos 2 variantes con el fin de eliminarla. Del análisis de ambas variantes pudimos ver que ninguna mejoraba sino que por el contrario empeoraban los resultados obtenidos hasta el momento. Por lo tanto dichas variantes fueron descartadas.

Posteriormente utilizamos 3 familias de desigualdades válidas, analizándolas por separado y también complementándose entre ellas. En las pruebas realizadas, vimos que la mejor combinación era utilizar en conjunto cortes Capacidad y Star y en los casos en que no se encontraran cortes de dichas familias agregar cortes Comb.

Finalmente, se realizaron comparaciones entre el algoritmo Branch-and-Cut provisto por CPLEX y nuestro algoritmo BC-V. En esta comparación se obtuvieron resultados notoriamente favorables para el algoritmo BC-V basándose en los tiempos de ejecución, gaps entre cotas inferiores y superiores y cantidad de nodos recorridos.

Como posible trabajo futuro podemos mencionar los siguientes puntos:

- Para la detección de ciclos además de las variantes de fuerza bruta, se podría considerar el desarrollo de un algoritmo de recorrido DFS sobre el grafo inducido por la solución.

- Nuestra heurística inicial proporciona una cota superior inicial adecuada pero de todos modos sería interesante desarrollar otro algoritmo que mejore aún más dicha cota sin invertir un tiempo excesivo en la ejecución.
- Si bien nuestros criterios de rompimiento de simetría no mejoraron la performance general, creemos que sería importante invertir tiempo en el desarrollo de otras propuestas ya que suponemos que la mejora que se obtendría sería importante. Para esto deberían buscarse nuevas condiciones para romper la simetría con la idea de que las desigualdades generadas agreguen la menor complejidad posible al problema. Una opción posible podría ser tener en cuenta las distancias para llegar y volver a las casas de los visitantes multiplicando dichas distancias por un epsilon chico.
- Además de las 3 familias de cortes implementadas podrían incorporarse nuevas familias y a partir de los resultados obtenidos de las ejecuciones individuales, realizar combinaciones de familias similares a las ya realizadas.

Otros temas a considerar a futuro son:

- Estudiar el comportamiento del algoritmo modificando los valores de alfa y beta presentes en la función objetivo.
- El modelo propuesto es fácilmente adaptable a problemas más generales, donde los clientes y los visitantes pueden tener características no homogéneas. Por ejemplo que no todos los visitantes trabajen la misma cantidad de horas y por lo tanto no todos tengan asignada la misma cantidad de clientes. También podríamos tener clientes que sólo pueden ser atendidos por algunos visitantes determinados para cada uno de ellos.

Bibliografía

- [1] J. Araque, L. Hall and T. Magnanti. *Capacitated trees, capacitated routing and associated polyhedra*, Discussion paper, Center for Operations Research and Econometrics, Catholic University of Louvain, Belgium, 1990.
- [2] P. Augerat, J. Belenguer, E. Benavent, A. Corberán, D. Naddef and G. Rinaldi, G. *Computational results with a branch-and-cut code for the capacitated vehicle routing problem*, Research report RR949-M, ARTEMIS-IMAG, France, 1995.
- [3] J. Beasley. *Lagrangian relaxation, ch. 6*, Modern Heuristic Techniques for Combinatorial Problems, Blackwell Scientific Publications, 1993.
- [4] R. Gomory. *Outline of an algorithm for integer solution to linear programs*, Bulletin American Mathematical Society 64, 275-278, 1958.
- [5] M. Grötschel, L. Lovasz and A. Schrijver. *The ellipsoid method and its consequences in combinatorial optimization*, Combinatorica 1, 169-197, 1981.
- [6] M. Grötschel and M. Padberg. *On the Symmetric Traveling Salesman Problem I: inequalities*, Mathematical Programming 16, 265-280, 1979.
- [7] A. Letchford, R. Eglese and J. Lysgaard. *Multistars, partial multistars, and the capacitated vehicle routing problem*, Mathematical Program. 94(1), 21-40, 2002.
- [8] J. Lysgaard, A. Letchford and R. Eglese, *A New Branch-and-Cut Algorithm for the Capacitated Vehicle Routing Problem*. Mathematical Programming 100(2), 423-445, 2004.
- [9] Bazaraa, M., Jarvis, J., *Linear Programming and Network Flows*. John Willey & Sons, 1977.
- [10] Chvátal, V., *Linear Programming*. Freeman, 1983.
- [11] Nemhauser, G., Wolsey, L., *Integer and Combinatorial Optimization*. John Willey & Sons, 1988.
- [12] Papadimitriou, C., Steiglitz, K., *Combinatorial Optimization*. Dover, 1998.
- [13] Schrijver, A., *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

- [14] Wolsey,L., *Integer Programming*. John Willey & Sons, 1998.