BCP
Una biblioteca para el desarrollo de algoritmos tipo Branch-and-Cut-and-Price

BCP: A library for Branch-and-Cut-and-Price algorithm development

Kenny Angelo Martín Rodríguez
Departamento de Ingeniería Informática y Sistemas
Escuela Técnica Superior de Ingeniería Informática

Trabajo de Fin de Grado

La Laguna, 8 de julio de 2014

D. **Jorge Riera-Ledesma**, con N.I.F. 43788061-V, profesor adscrito al Departamento de **Ingeniería Informática y Sistemas** de la Universidad de La Laguna

C E R T I F I C A

Que la presente memoria titulada:

"BCP: una librería para el desarrollo de algoritmos tipo Branch-and-Cut-and-Price."

ha sido realizada bajo su dirección por D. **Kenny Angelo Martín Rodríguez**, con N.I.F. 42418153-N.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos, firman la presente en La Laguna a 10 de junio de 2014.

Fdo: D. **Kenny Angelo Martín Rodríguez** Fdo: D. **Jorge Riera-Ledesma**

Agradecimientos

Para empezar, me gustaría agradecer a mi familia por haberme apoyado tanto y confiar ciegamente en mí, a pesar de todas las adversidades que han podido surgir. También quería agradecer a mis amigos, tanto a los ya conocidos como a los que he ido conociendo a lo largo de mi carrera, por estar ahí en los malos momentos y en los peores, aún más.

Por otro lado, quisiera reconocer la educación de D. Jorge Riera-Ledesma con la que se ha dirigido a mí en todo el periodo de la realización del Trabajo de Fin de Grado. Así como su capacidad de ver todo desde un lado positivo y calmado.

Asimismo, me gustaría dar las gracias a todas las personas, tanto profesores como alumnos de la Universidad de La Laguna, e incluso ajenos a la Universidad. Independientemente de si hemos compartido una relación positiva, gracias a todos ellos, porque eso me ha ayudado a crecer como persona, a volverme cada vez más fuerte y afrontar todos los problemas que se han ido interponiendo en mi camino de forma persistente.

Por último, quiero dedicar la realización de este proyecto a mi madre, una persona guerrera con un espíritu de superación increíble, que ha sido mi referente cada día para luchar y conseguir los objetivos en mi vida. Gracias a ella he conseguido todo lo que tengo hasta ahora y, por ello, se merece estas palabras y mucho más.

Resumen

Las bibliotecas (del inglés library) son un conjunto de funciones codificadas en un lenguaje de programación determinado que ofrece una interfaz bien definida para su uso.

El objetivo de este trabajo ha sido el crear una biblioteca de clases en C++ que implementa un algoritmo de tipo Branch-and-Cut-and-Price. Este tipo de algoritmos utiliza técnicas de descomposición – como la de Benders o Dantzig-Wolfe – para mejorar los algoritmos y ofrecer soluciones a problemas de gran tamaño. Se pretende que nuestra biblioteca se componga (esté compuesta por) de un conjunto de objetos adaptables, de forma que los métodos de las clases puedan ser modificados o enriquecidos para adaptarse a las especificaciones de cada problema.

Palabras clave

Biblioteca, Branch-and-Bound, Branch-and-Cut, Branch-and-Price, Branch-and-Cut-and-Price, descomposición Dantzig-Wolfe, problema de optimización, Programación Lineal Entera Mixta

Abstract

A library is a collection of functions, written in terms of a programming language, which has a well-defined interface so that we can use them.

The main goal of this project is to create a library in C++, which implements a Branch-and-Cut-and-Price algorithm. This kind of algorithms makes use of decomposition techniques – such as Benders or Dantzig-Wolfe – in order to improve algorithm performance and offer solutions for large problems. We expect our library to consist of a bunch of adaptable objects in a way that class methods could be modified or enriched to fit the requirements of each problem.

Keywords

Library, Branch-and-Bound, Branch-and-Cut, Branch-and-Price, Branch-and-Cut-and-Price, Dantzig-Wolfe decomposition, optimization problem, Mixed Integer Linear Programming

Índice general

Capítulo I. Introducción

1. Introducción

1.1 Descripción del proyecto

Los problemas de *Programación Lineal Entera* complejos requieren en ocasiones de una gran cantidad de recursos computacionales para su resolución. Por ello, suelen ser abordados a través de transformaciones que permitan descomponer el problema en subproblemas más pequeños. Aunque la resolución del problema principal implica también la resolución de los subproblemas, estos últimos requieren menos recursos.

Una de estas transformaciones se conoce como *descomposición de Dantzig-Wolfe*. Esta transformación elimina un conjunto de restricciones del programa lineal, y las transforma en columnas o variables. Sin embargo, el número de columnas obtenidas crece de forma exponencial con respecto al número de restricciones eliminadas. La inclusión de todas las columnas haría la resolución del problema inabordable. Por ello, las columnas se introducen de forma dinámica en el programa lineal a medida que éstas son requeridas por el resolutor lineal. La generación lineal de columnas se conoce como *Pricing*.

Cuando las variables originales del problema son enteras, la resolución del problema demanda la utilización de un árbol de ramificación que divida el espacio de decisión de acuerdo con los posibles valores de las variables. La división del espacio da lugar a un árbol de decisión con un número exponencial de ramas con respecto al número de variables, así que se deben establecer criterios para eliminar aquellas ramas que conduzcan a soluciones no prometedoras. La utilización de un árbol de ramificación para la resolución de un problema de Programación Lineal Entera se denomina *Branch-and-Bound*.

La técnica que combina la generación dinámica de columnas con un árbol de ramificación se conoce como *Branch-and-Price*. Si además de generar columnas, se generan restricciones – también dinámicamente – dicha técnica se denomina *Branch-and-Cut-and-Price*.

El manejo dinámico de variables, así como la gestión del árbol de ramificación, puede llevarse a cabo de forma automatizada en un entorno de resolución de problemas de Programación Lineal Entera. El objeto de este proyecto es la elaboración de una biblioteca de clases, escrito en C++, que implemente un algoritmo de tipo Branch-and-Cut-and-Price, facilitando la resolución de estos problemas de *Programación Lineal Entera Mixta* mediante el método de descomposición de Dantzig-Wolfe.

1.2 Antecedentes

El desarrollo de bibliotecas para la resolución de problemas lineales mediante las técnicas Branch-and-Cut, Branch-and-Price y Branch-and-Cut-and-Price ha sido el objeto de trabajo de diversos proyectos. Entre ellos, cabe destacar el proyecto COIN-OR [1], desarrollado por International Business Machines y que se basa en *COIN-OR LP Solver*: un resolutor lineal; *Open Solver Interface*: una interfaz que independiza la

resolución de los problemas de su optimizador línea; y *Branch-and-Cut-and-Price*: una biblioteca que reúne herramientas de programación para la resolución de problemas mediante esta técnica.

Otra herramienta similar es *Solving Contraint Integer Programs* [2], desarrollada por el Zuse Institute Berlin, que consiste en una biblioteca y un marco de trabajo para la resolución de programas lineales enteros.

Estas dos bibliotecas han sido sometidas a diversas evaluaciones a través de programas de ejemplo y han superado las expectativas de calidad. La herramienta que se pretende desarrollar persigue un objetivo similar. Aunque, en lugar de intentar conseguir un producto final, se pretende que se introduzcan las técnicas de resolución y se comprendan a través de la implementación de esta biblioteca.

1.3 Programación Lineal

La *Programación Lineal* [3] es una técnica matemática surgida en el siglo XX. Consiste en una serie de métodos y procedimientos que permiten modelar y resolver problemas de optimización, cuya característica principal es maximizar o minimizar una función lineal – llamada función objetivo – y cuyas variables se encuentran sujetas a una serie de restricciones; dichas restricciones serán representadas por un sistema de inecuaciones lineales.

1.3.1 Programación Lineal Entera

En algunos casos, se necesita que la solución óptima, es decir, aquella que maximice – o minimice, según convenga – el valor objetivo, esté compuesta por algunas variables restringidas a tomar valores enteros. Se habla entonces de *Programación Lineal Entera* [4].

Estos problemas se formulan de la misma manera que los de Programación Lineal, pero agregando la condición de que al menos una de las variables tomará valores enteros. Analizando las posibles alternativas de que estas variables tomen un valor entero u otro en un entorno, a partir de una solución obtenida considerando las variables reales, podemos resolver este problema.

Los problemas de Programación Lineal Entera van a permitir modelar muchos más casos que los de Programación Lineal, pero con la contrapartida de suponer un coste computacional mucho mayor para su resolución.

La causa de este incremento de cómputo se debe a que desaparece la propiedad que poseen los problemas de Programación Lineal de que al menos una solución óptima del problema se encuentra en un punto extremo. En los problemas de Programación Lineal Entera, los conjuntos pueden estar definidos a trozos y ser menos convexos, con lo que la idea de punto extremo se descarta.

Por todo ello, se hace necesario usar algún método de diseño de algoritmos, como la técnica Branch-and-Bound [5], que parte de añadir nuevas restricciones para cada variable de forma que, cuando vuelva a ser evaluado, lleve al óptimo entero.

1.3.2 Programación Lineal Entera Mixta

Ya se ha visto que la Programación Lineal Entera define aquel modelo donde las variables son números enteros no negativos. En situaciones reales, se puede hacer frente a problemas de decisión (es decir, decisiones de sí o no), que pueden representarse con variables denominadas binarias. Cuando sólo es necesario que algunas de las variables sean enteras y el resto continuas, el modelo recibe el nombre de problema de *Programación Lineal Entera Mixta* [6]; esta clasificación incluye también aquellos modelos que se compongan tanto de variables enteras no negativas y continuas, como de variables binarias.

1.4 Branch-and-Bound

El *método de Branch-and-Bound* es una variante de la *técnica de Backtracking*, mejorado sustancialmente pare resolver problemas de optimización.

Se interpreta como un árbol de soluciones donde el recorrido no tiene por qué ser en profundidad – como ocurría en backtracking – sino que se sigue una estrategia de ramificación. Además, se utilizarán técnicas de poda para detectar en qué ramificación ya no son óptimas las soluciones dadas (a través de estimaciones de cotas para cada nodo). En caso de no ser óptima, dicha ramificación se podará para no malgastar recursos y procesos en casos que se alejan de la solución óptima.

El algoritmo de Branch-and-Bound sigue, básicamente, el procedimiento siguiente:

- Se introducen los hijos del nodo raíz en una *Lista de Nodos Vivos* (LNV), que contendrá los nodos pendientes de expandir por el algoritmo. Según como estén almacenados los nodos en la LNV, el recorrido se realizará de una forma u otra, dando lugar a la estrategia de ramificación.
- Se extrae uno nodo de la LNV.
- Vemos si el nodo es factible (propone una solución válida) y óptimo (se compara con la mejor solución encontrada de momento).
- Si ese nodo es solución, se almacena; si no lo es, se introduce en la lista los hijos de dicho nodo.

1.4.1 Aplicación en problemas de Programación Lineal Entera Mixta

A continuación, se explicará el funcionamiento de la técnica Branch-and-Bound enfocada a problemas MILP (*Mixed Integer Linear Programs* [6]), partiendo del algoritmo básico y utilizando técnicas avanzadas para su resolución.

El algoritmo de Branch-and-Bound resuelve un MILP dividiendo el espacio de búsqueda y generando una secuencia de subproblemas; este espacio de búsqueda se puede representar por un árbol. Cada nodo del árbol corresponde a un subproblema que deriva de los problemas previos a éste y que se encuentran en el camino que lleva del

nodo en cuestión a la raíz del árbol. El subproblema ($MILP^0$) asociado a la raíz es idéntico al problema original, al cual llamamos *MILP*.

La *relajación lineal* (LP^0) de $MILP^0$ puede ser definida como:

(1)

$$\min_x c^T x$$
$$s.a.: Ax \leq b$$
$$l \leq x \leq u$$

El algoritmo genera subproblemas a lo largo del árbol usando el siguiente esquema. Considérese $\bar{x}^0$, la solución óptima de LP^0, que suele calcularse utilizando el algoritmo simplex dual. Si $\bar{x}_i^0$ es entero $\forall i \in \ell$, entonces $\bar{x}^0$ es una solución óptima para *MILP*. Supóngase que $\exists i \in \ell$, tal que $\bar{x}_i^0$ no es entero. En tal caso, el algoritmo define dos nuevos subproblemas ($MILP^1$ y $MILP^2$), descendientes del nodo padre $MILP^0$. El subproblema $MILP^1$ es idéntico al padre, excepto por la restricción adicional

(2)

$$x_i \leq \lfloor \bar{x}_i^0 \rfloor,$$

y el subproblema MILP^2 es idéntico también, exceptuando la restricción adicional

(3)

$$x_i \geq \lceil \bar{x}_i^0 \rceil.$$

La notación $\lfloor y \rfloor$ representa el entero más grande que es menor o igual a *y*, y la notación $\lceil y \rceil$, el entero más pequeño que es mayor o igual a *y*. Ambas restricciones pueden ser manejadas modificando las cotas de la variable x_i, en lugar de añadir explícitamente las restricciones a la matriz. Los dos nuevos subproblemas no tienen a $\bar{x}^0$ como solución factible, pero la solución entera del problema *MILP* debe satisfacer una de las dos restricciones (la de $MILP^1$ o $MILP^2$). Por tanto, ambos subproblemas se definen como nodos activos del árbol de Branch-and-Bound y la variable x_i se denomina variable de ramificación.

En el siguiente paso, el algoritmo escoge uno de los nodos activos e intenta resolver la relajación lineal asociada a dicho subproblema. La relajación podría ser no factible, en cuyo caso el subproblema se descarta. En cambio, si puede ser resuelto y la solución es factible entera (es decir, x_i es entero $\forall i \in \ell$), entonces el valor objetivo proporciona la cota superior para el problema de minimización *MILP*; si la solución no es factible entera, nuevamente se definen dos nuevos subproblemas. Este proceso continúa iterando hasta que ya no haya nodos activos. Llegados a este punto, la mejor solución entera encontrada es la solución óptima para el problema *MILP*; si no se ha encontrado una solución entera, entonces se dice que el problema *MILP* no tiene una solución entera factible.

1.4.2 Branch-and-Cut y su aplicación en problemas de MILP

En este apartado se hablará de la *técnica de Branch-and-Cut* [7], que se basa en el algoritmo de Branch-and-Bound con la inclusión de planos de corte. Así, se describe una técnica que nos permite obtener óptimas para problemas formulados mediante un modelo de Programación Lineal Entera, usando algoritmos simplificados. Se puede obtener una descripción detallada de las técnicas en libros de autores como Jünger y Thienel [8], Padberg y Rinaldi [9] o Thienel [10].

La relajación lineal de un problema lineal entero (IP) es el problema lineal obtenido del IP, eliminando las restricciones de integralidad. Por lo tanto, el valor óptimo de la relajación w^{LP} en el caso de minimización es una cota inferior del valor óptimo w^{IP} del problema lineal entero. O sea, $w^{LP} \leq w^{IP}$.

La técnica Branch-and-Cut es una metodología que permite resolver problemas lineales con un número exponencial o, al menos, muy grande de restricciones. Estas restricciones generalmente vienen dadas de forma implícita. Véase la Figura 1 para una ilustración de esta técnica.

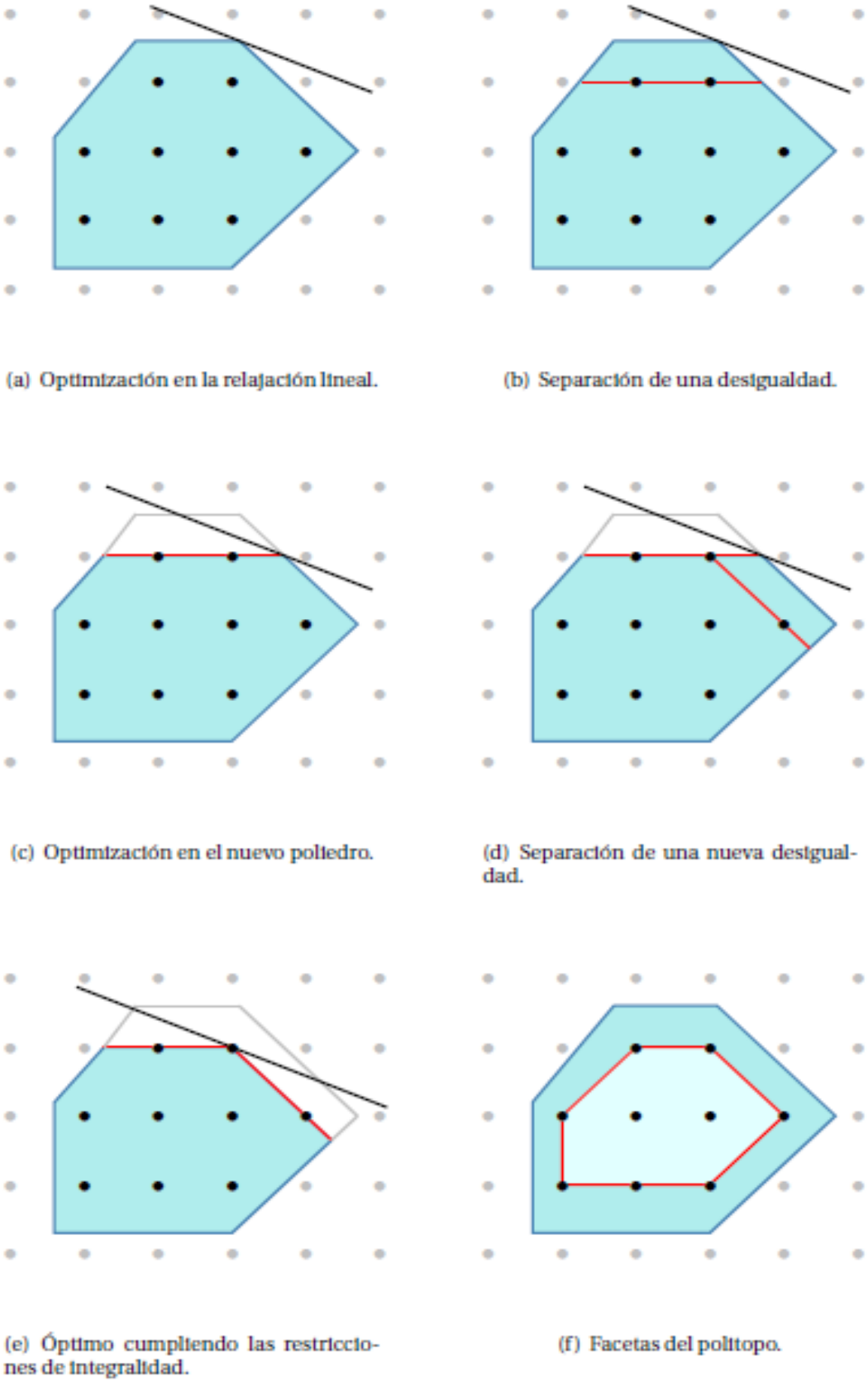
Denotemos por $\prod$ un problema de optimización genérico. Este problema es una estructura de datos que consta de una lista de restricciones activas L^C, la lista de variables activas L^V, el vector de optimización w, el vector solución de la relajación lineal actual z^*, el valor objetivo de la relajación lineal actual w^{LP} de acuerdo con w y el estado de la relajación lineal.

En un esquema básico de esta técnica para un problema de minimización, una lista L^P de subproblemas es inicializada con la relajación lineal del problema $\prod$. El valor de la mejor solución z encontrada hasta el momento es almacenado como cota superior global w^{UB}. Cada iteración selecciona un subproblema de la lista de problemas pendientes. La cota inferior es calculada para cada subproblema resolviendo la relajación lineal con el conjunto actual de restricciones activas y variables activas. Por otro lado, el algoritmo intenta mejorar la cota superior mediante una heurística basada en la solución obtenida en la relajación lineal actual. El cálculo de esta cota ofrece una garantía de la distancia a la optimalidad, es decir, para problemas de gran tamaño – y/o duros – el algoritmo de Branch-and-Bound permite obtener una buena solución (posiblemente óptima) e indica cuán lejos está la solución de llegar a ser óptima.

Un subproblema es podado de la lista si se cumple alguna de las siguientes condiciones:

- se obtiene una solución factible entera a partir de la relajación lineal
- la relajación lineal es no factible
- la cota inferior local es mayor que la cota superior global.

Si la relajación lineal actual no fuera factible, entonces se lleva a cabo un intento por generar nuevas desigualdades válidas violadas por la solución fraccional actual, utilizando procedimientos de separación. Si se encuentra esa desigualdad, se añade al programa lineal, esperando que nos lleve a una solución diferente menos fraccional. Esto se repite hasta que se encuentra una solución entera o no se encuentran más planos de corte.



(a) Optimización en la relajación lineal.

(b) Separación de una desigualdad.

(c) Optimización en el nuevo poliedro.

(d) Separación de una nueva desigualdad.

(e) Óptimo cumpliendo las restricciones de integralidad.

(f) Facetas del politopo.

Figura 1 Técnica de Branch-and-Cut

1.5 Branch-and-Price

La *técnica de Branch-and-Price* es un método híbrido entre Branch-and-Bound y el *método de Generación de Columnas* [11]. Para cada nodo en el árbol de búsqueda, se pueden añadir columnas a la relajación lineal. Al comienzo del algoritmo, se excluye un conjunto de columnas de la relajación para reducir la complejidad computacional y los requisitos de memoria; luego, las columnas se añaden a la relajación conforme se vayan necesitando. Esto se hace posible ya que, en problemas de gran tamaño, la mayor parte de las columnas tendrán un valor igual a cero en cualquier solución óptima. En consecuencia, la inmensa mayoría de las columnas son irrelevantes y, por tanto, descartables para la resolución del problema.

1.5.1 Branch-and-Cut-and-Price y su aplicación en problemas de MILP

En este apartado se hablará de la *técnica de Branch-and-Cut-and-Price* para la resolución de problemas de MILP. En muchos problemas MILP, suele haber algún tipo de estructura asociada que pueda ser explotada normalmente por descomposición. Ya se ha visto en apartados anteriores que cuando la relajación en cada nodo del árbol de Branch-and-Bound es resuelta por generación de columnas, se habla de Branch-and-Price. Opcionalmente, se pueden añadir métodos de planos de corte para reforzar la relajación lineal; así surge la técnica de Branch-and-Cut-and-Price.

Se hace necesaria la reformulación del problema utilizando alguna técnica de descomposición, en este caso la técnica de descomposición de Dantzig-Wolfe. La relación entre el problema original y las reformulaciones permite entender cómo deben ser formulados los planos de corte y cómo deben tomarse las decisiones de ramificación, a la vez que mantenemos los problemas de generación de columnas manejables.

La descomposición y reformulación de programas enteros mixtos [12] supone un enfoque típico para obtener relajaciones más fuertes y reducir la simetría. Esto suele conllevar la adición dinámica de variables (columnas) y/o restricciones (planos de corte) al modelo. Por trivial que parezca esta técnica, hay que tener claro que tanto reforzar la relajación a través del método de planos de corte como realizar una ramificación sobre variables fraccionales interfieren en la generación de columnas y puede hacer desaparecer las ventajas de la descomposición si no se procede con precaución.

1.6 Métodos de descomposición de Dantzig-Wolfe

Una vez descritos los diferentes algoritmos utilizados, procederemos a la descripción del método de descomposición de Dantzig-Wolfe [13].

La motivación que lleva a utilizar métodos de descomposición surge cuando el tamaño del problema, en término de número de variables y restricciones, hace su resolución muy *pesada*. Para problemas pequeños o de tamaño medio, se pueden utilizar técnicas convencionales de Programación Lineal; pero resulta imprescindible mejorar la técnica utilizada en caso de que se quiera resolver problemas de grandes dimensiones. A

continuación, se describirá brevemente la técnica de la descomposición de Dantzig-Wolfe.

1.6.1 Descripción matemática

A continuación, se procederá a la descripción matemática del Método de Descomposición de Dantzig-Wolfe [13]. Se ha extraído del manual sobre Programación Matemática [14]. Supóngase el problema lineal:

(4)

$$\begin{aligned} (P)\ \ & \min_{x} c^T x \\ s.a.:\ & Ax = b \\ & x \geq 0 \end{aligned}$$

donde A y b son divisibles en la forma:

(5)

$$A = \begin{bmatrix} A^0 \\ A^1 \end{bmatrix}, b = \begin{bmatrix} b^0 \\ b^1 \end{bmatrix},$$

entonces el problema P puede verse como:

(6)

$$\begin{aligned} (P)\ \ & \min_{x} c^T x \\ s.a.:\ & A^0 x = b^0 \\ & A^1 x = b^1 \\ & x \geq 0. \end{aligned}$$

Considérese el poliedro $Q_1 = \{x\colon A^1 x = b^1, x \geq 0\}$, y asúmase que $\{p^1, \dots, p^p\}$ son sus puntos extremos y $\{d^1, \dots, d^q\}$, sus direcciones extremas. Se asume también que $p + q \geq 1$, ya que de lo contrario P es no factible. Entonces, en virtud del teorema de Caratheódory [14], cualquier punto $x \in Q_1$ (si existe alguno) es identificable con valores $\lambda_i (i = 1, \dots p)$ y $\mu_j\ (j = 1, \dots, q)$ tales que $x = \sum_{i=1}^{p} \lambda_i p^i + \sum_{j=1}^{q} \mu_j d^j, \lambda_i \geq 0\ (i = 1, \dots, p)$ y $\mu_j \geq 0\ (j = 1, \dots, q)$, y $\sum_{i=1}^{p} \lambda_i = 1$. Por tal caracterización, el problema P es también reformulable como:

(7)

$$(P)\ \min_{\lambda_i, \mu_j} \sum_{i=1}^{p} \lambda_i c^T p^i + \sum_{j=1}^{q} \mu_j c^T d^j$$

$$s.a.: \sum_{i=1}^{p} \lambda_i A^0 p^i + \sum_{j=1}^{q} \mu_j A^0 d^j = b^0$$

$$\sum_{i=1}^{p} \lambda_i = 1$$

$$\lambda_i \geq 0 \qquad \forall i = 1, \dots, p$$

$$\mu_j \geq 0 \qquad \forall j = 1, \dots, q$$

Esta reformulación del problema P es conocida como *problema maestro* (completo). Dado que los problemas P y P^M son equivalentes, ¿cuál es la ventaja de tal reformulación? Al pasar desde P hasta P^M las filas $A^1 x = b^1$ se han sustituido por una única $\sum_{i=1}^{p} \lambda_i = 1$ (fila de convexidad), a costa de aumentar el número de columnas hasta otro número extremadamente enorme. Debemos notar sin embargo que, usando el *simplex revisado* para la resolución del problema maestro, no es necesario disponer de todas las columnas en cada iteración. Véase ahora como proceder a la generación de columnas (puntos/direcciones extremos de Q_1).

Supóngase que se conocen algunas columnas γ de P^M, es decir, que se dispone del *problema maestro relajado* $P^M(\gamma)$. Sin pérdida de generalidad, se puede asumir que estos problemas maestros relajados son siempre factibles, añadiendo una variable artificial con costo muy grande; si al final del proceso se observa que el valor objetivo óptimo del último $P^M(\gamma)$ depende de este costo muy grande, entonces P es no factible. Si al resolver $P^M(\gamma)$ resulta no acotado, entonces el problema maestro completo (y en consecuencia P) es no acotado (basta ampliar la dirección de descenso para $P^M(\gamma)$ con ceros en las columnas en γ para crear una dirección de descenso para P^M). En otro caso, sea (u, α) un vector de variables duales óptimo para $P^M(\gamma)$ (las componentes del vector u están asociadas a las filas de $\sum_{i=1}^{p} A^0 p^i \lambda_i + \sum_{j=1}^{pq} A^0 d^j \mu_j = b^0$ y α a la fila de convexidad). Entonces, usando las condiciones de holguras complementarias, el vector (u, α) será una solución dual óptima para P^M cuando verifique:

(8)

$$u^T A^0 p^i + \alpha \leq c^T p^i \qquad \forall i = 1, \dots, p$$

$$u^T A^0 d^j \leq c^T d^j \qquad \forall j = 1, \dots, q$$

esto es, cuando:

(9)

$$c^T p^i - u^T A^0 p^i - \alpha \geq 0 \qquad \forall i = 1, \dots, p$$

$$c^T d^j - u^T A^0 d^j \geq 0 \qquad \forall j = 1, \dots, q$$

Para estudiar esto basta resolver:

(10)

$$\min_i c^T p^i - u^T A^0 p^i - \alpha = \min_i \left[(c^T - u^T A^0) p^i \right] - \alpha$$

$$\min_j c^T d^j - u^T A^0 d^j = \min_j \left[(c^T - u^T A^0) d^j \right]$$

Tras ello, hay que ver si tienen un valor objetivo óptimo no negativo. En tal caso, el método del simplex revisado sobre P^M ya puede detenerse porque tiene su solución

óptima; en caso contrario, se tiene una variable p^i o d^j que conviene que entre en la base actual, y se prosigue haciendo iteraciones con el simplex revisado. Nótese finalmente que la resolución de los dos problemas anteriores queda resuelta aplicando el método del simplex al siguiente problema, conocido como *subproblema de P*:

(11)

$$(P^S) \quad \min_x (c^T - u^T A^0)x$$
$$s.a.: \ A^1 x = b^1$$
$$x \geq 0$$

Esto conduce a un algoritmo iterativo en el que, iteración tras iteración, se va enriqueciendo el problema máster con los cortes obtenidos de la resolución del subproblema.

1.6.2 Interpretación descriptiva

De forma resumida, se podría presentar el funcionamiento de este método de descomposición en una serie de pasos. Estos pasos no tienen por qué definirlo completamente, ya que hay variaciones de la técnica.

1. Empezamos con una solución factible del *problema maestro restringido*, formulando nuevas funciones objetivo para cada subproblema de forma que éstas ofrezcan una mejora del valor objetivo actual del programa maestro.
2. Dadas estas nuevas funciones objetivo, los subproblemas se vuelven a resolver. Por cada subproblema, el problema maestro recibe un valor óptimo.
3. El maestro incorpora una, o todas, las nuevas columnas generadas por las soluciones de los subproblemas, basándose en la habilidad de cada columna de mejorar el valor objetivo.
4. Se realizan tantas iteraciones del algoritmo simplex como nuevas columnas se hayan añadido.
5. Si la función objetivo mejora, se vuelve a repetir todo el proceso.
6. Criterio de parada: ocurre cuando ninguna nueva columna de los subproblemas mejora el valor objetivo del problema maestro.

Capítulo II. Descripción de la Biblioteca

2. Descripción de la Biblioteca

Ya se ha hablado de las nociones teóricas necesarias para entender el funcionamiento de la biblioteca que se pretende realizar. Por tanto, el siguiente paso es describir el modelo concreto de dicha biblioteca, incluyendo su estructura, así como el funcionamiento de cada parte que la compone.

2.1 Estructura

Nuestra biblioteca se compone de cinco partes principales:

- *Descriptor de variables y restricciones,* el cual definirá la familia de variables que van a intervenir en el problema y, por ende, se encargará de establecer las restricciones y el problema en general.
- *Árbol de búsqueda,* que se encargará de gestionar los nodos.
- *Nodo*, empezando por el nodo raíz y que, junto al resto, conforman el árbol de búsqueda. Tendrá la configuración del problema de MILP asociado, específico para cada nodo.
- *Separador*, que se encargará de añadir dinámicamente, si fuera preciso, las filas necesarias – ó también llamadas restricciones – al nodo que tiene como solución del problema de MILP asociado, una solución fraccionaria.
- *Pricer*, análogo al separador, introducirá columnas adicionales a través de un algoritmo de generación dinámica de columnas.

2.2 Funcionamiento

2.2.1 Descriptor de variables y restricciones

Como se ha citado previamente, el descriptor de variables define el modelo del problema que se va a abordar, estableciendo el conjunto de variables que se pretende usar y las restricciones asociadas a éstas. Para poder explicar el funcionamiento de la biblioteca que se ha diseñado, es imprescindible tener un modelo de ejemplo que, en resumidas cuentas, será traducido en líneas de código. De esta manera, tenemos el siguiente modelo:

(12)

$$s_j - C^* \leq 0 \quad j \in J$$

(13)

$$e_j^k - s_j \leq 0 \qquad j \in J, k \in K$$

(14)

$$-e_j^k - w_j^k + s_j \leq 0 \qquad j \in J, k \in K$$

(15)

$$\sum_{k \in K} \sum_{i \in V} x_{ij}^k = q_j \qquad j \in J$$

(16)

$$\sum_{i \in V} x_{0i}^k = 1 \qquad k \in K$$

(17)

$$0 \leq x_{ij}^k \leq 1 \qquad (i,j) \in A, k \in K$$

(18)

$$x_{ij}^k - \sum_{p \in \Omega_k(C^*)} x_{ij}^p \lambda^p = 0 \qquad (i,j) \in A, k \in K$$

(19)

$$s_j^k - \sum_{p \in \Omega_k(C^*)} s_j^p \lambda^p = 0 \qquad j \in J, k \in K$$

(20)

$$\lambda^p \geq 0 \qquad p \in \Omega_k(C^*), k \in K$$

En dicho modelo, hay dos familias de variables: las variables del problema Maestro y las variables del subproblema. El conjunto de variables x y s, que corresponden a las restricciones (18) y (19), pertenecen al subproblema; mientras que el resto es parte del problema Maestro.

El fichero "variables.cc" (Véase Apéndice A.1) describe cómo introducir las variables del modelo utilizando la biblioteca BCP. El procedimiento consiste simplemente en rellenar los elementos de un array de objetos $VarDes\_$ (que hace referencia a la descripción de variables). Estas variables se introducen en un vector, insertando primero el conjunto de variables x, después el conjunto e y, por último, el conjunto s.

En los campos del objeto se introduce los siguientes valores:

- *vt* = tipo de variable, que puede ser binaria (BCP_template::_BINARY_) o continua (BCP_template::_CONTINUOUS_)
- *st* = indica si la variable es del problema Maestro (BCP_template::_MASTER_PROBLEM_) o del subproblema (BCP_template::_SUBPROBLEM_)
- *c* = valor objetivo de la variable
- *lb* = cota inferior de la variable
- *ub* = cota superior de la variable

Por otro lado, el fichero "restricciones.cc" (Véase Apéndice A.2) corresponde con la introducción de un conjunto de restricciones que conforma el problema. Se introduce cinco familias de restricciones, separadas por comentarios. Cada una viene descrita por un vector disperso compuesto de componentes $column$, que hace referencia al índice de la variable que forma parte de la restricción, y $values$, que representa el coeficiente de cada uno de estos índices. Una vez descrita, se crea la fila con la información asociada y se añade al pool de restricciones.

2.2.2 Árbol de búsqueda

La clase *BCP_branching_tree* representa un árbol de ramificación. Dicho árbol, optimiza cada nodo, introduce de forma dinámica restricciones y fija en cada uno de sus nodos una variable entera a su límite superior o inferior. El árbol de ramificación implementa recorridos en amplitud, profundidad, y recorrido Best Bound. El criterio de selección de variable para realizar la ramificación se basa en aquella más próxima a 0.5 en el caso de variables binarias.

No obstante, a pesar de contar con todas estas estrategias de base, el usuario de la clase puede redefinir cada procedimiento a fin de confeccionar las estrategias que mejor se adapten al problema específico que se esté abordando.

Los archivos "BCP_branching_tree_t.hpp" (Véase Apéndice A.3) y "BCP_branching_tree_t.cpp" (Véase Apéndice A.4) representan la clase gestora del árbol de ramificación. Esta clase dispone de una lista enlazada de nodos que almacena tanto los elementos por procesar ($tree\_$), como la lista de los ya procesados ($already\_processed$).

El procedimiento *process_tree* extrae un elemento de la lista de nodos pendientes de ser procesados e invoca al procedimiento *optimize* del nodo. Dicho procedimiento resuelve el programa lineal e introduce las filas y columnas que sean necesarias.

Una vez hecho esto, se comprueba si la cota inferior obtenida es inferior a la mejor cota superior guardada hasta el momento. Si es así y se encuentra alguna variable fraccionaria, se invoca al procedimiento *do_branch*, que se encarga de seleccionar la variable por la cual ramificar. Esta ramificación consiste en crear dos nodos a partir de la variable fraccionaria e introducir ambos en $tree\_$. A continuación, se almacena el nodo ya procesado en la lista de nodos procesados.

2.2.3 Nodo

Se trata de una de las clases más importantes, puesto que gestiona las operaciones del nodo en el árbol de ramificación. La clase *BCP_branching_node* representa cada subproblema asociado a un programa lineal entero mixto con alguna de estas variables enteras fijadas. La descripción del programa lineal se encuentra dentro un atributo del tipo *BCP_node_solver*.

La fijación de variables se consigue cuando, durante el proceso de ramificación, el método *do-branch* de la clase *BCP_branching_tree* crea dos nodos, subdividiendo el

espacio de solución, e introduce una restricción sobre la variable fijada en cada uno de los nodos resultantes.

La clase BCP_branching_node dispone de un método de activación. El propósito de este método es reconstruir y activar un programa lineal a partir de la información guardada de forma comprimida por el árbol de ramificación correspondiente al programa lineal de su nodo padre. Una vez reconstruido el nodo, se añade la restricción que se ha introducido para fijar la variable asociada a dicho nodo.

Finalizada la activación, la clase gestora del árbol invoca el método optimización. Este método resuelve el programa lineal a través de un resolutor linear comercial; en esta primera versión sólo es posible utilizar CPLEX. Una vez hecho esto, se invoca al método de generación de filas (restricciones) y el de generación de columnas.

Se puede entender esta descripción con el archivo "BCP_branching_node_t.hpp" (Véase Apéndice A.5) y el archivo "BCP_branching_node_t.cpp" (Véase Apéndice A.6). Entre los atributos reseñables se encuentra:

- Un puntero a la estructura *dualSolution_t* dualSolution_*, en la que se almacena la solución del programa lineal, en otras palabras, almacena el valor de las variables primales, duales y de los costos reducidos de las mismas.
- El atributo *nodeSolver_t* nodesolver_* almacena un puntero al resolutor lineal (CPLEX, como antes se citaba).
- El atributo *branching_node_t** children_* es un array con los nodos hijos (que suelen ser 2).
- *basic_colDataNodePool_t nodeCols_* es el conjunto de columnas utilizadas en este nodo.
- *rowDataNodePool_t nodeRows_* es el conjunto de filas utilizadas en este nodo.
- *rowDataProblemPool_t& problemRows_* es el conjunto de filas generadas por todos los nodos en este momento.
- *basic_colDataProblemPool_t& problemCols_* es el conjunto de columnas generadas hasta el momento de forma global.
- *primal_heuristic_t& heuristic_* es una referencia al objeto que calcula las cotas superiores a partir de heurísticas definidas por el usuario.
- *statistics_t& statistics_* almacena las estadísticas del problema.
- *problemInfo_t& problemInfo_* almacena información específica del problema.
- *problemSolution_t& solution_* guarda la mejor solución factible encontrada hasta el momento.
- *pricer_t& pricer_* es un objeto generador de columnas.
- *separator_t& separator_* es un objeto generador de filas.

Por otro lado, en cuanto al procedimiento más importante que es *optimize*, se puede hacer una descripción más profunda:

1. Se resuelve el programa lineal. A partir de este método se actualiza la solución parcial obtenida hasta el momento.

2. Si el número de filas generadas es cero, entonces se invoca al generador de columnas, que las añadirá al problema lineal.
3. Análogamente, si el número de columnas generadas es cero, entonces se invoca el generador de filas; si se encuentras nuevas filas, se añaden al problema lineal.
4. El procedimiento continúa iterando mientras se encuentren nuevas filas o columnas que añadir.

2.2.4 Separador

En el transcurso de un proceso iterativo, se determina si es necesario introducir una restricción dinámica, mediante el algoritmo de separación específico que hay definido el usuario en la clase *BCP_separator*.

Este método lleva a cabo un algoritmo definido por el usuario de la biblioteca que, a partir de una solución fraccionaria, determina si es necesario introducir una restricción. Si así fuera, introduciría la restricción asociada a la solución parcial y resolvería nuevamente el programa lineal.

Análogo a la explicación de la clase nodo, se hablará a continuación en referencia a los archivos "BCP_branching_separator_t.hpp" (Véase Apéndice A.7) y "BCP_branching_separator_t.cpp" (Véase Apéndice A.8), para poder entender de una forma más clara la biblioteca.

Como se ha comentado, se trata de la clase generadora de restricciones, que deriva de la clase *separator_t.* En ella se debe implementar dos métodos virtuales:

- El método *separate*, que debe encontrar restricciones violadas por una solución descrita por el vector x e introducirlas en el pool de restricciones (row pool). El contenido del método no es relevante en sí, pues depende enormemente del problema que se quiera definir.
- El procedimiento *feasible* analizará una solución y determinará si hay alguna restricción violada; si no existiera ninguna violación de restricciones, entonces la solución será factible.

2.2.5 Pricer

Un proceso similar es efectuado por el módulo de generación de columnas. Una vez que se ha determinado que no es necesario introducir una nueva restricción se evalúa la posibilidad de introducir columnas adicionales a través de un algoritmo de generación de columnas o *pricing*. En tal caso se introducirían de forma dinámica, y, de forma similar al proceso de separación de restricciones, se resolvería nuevamente el programa lineal.

Este procedimiento iterativo de generación dinámica de filas y columnas se repite iterativamente hasta que no se detecte nuevas restricciones o nuevas columnas. Es entonces cuando se da por concluida la optimización del nodo, se actualizan las cotas

inferiores y superiores del mismo y se invoca al procedimiento de ramificación si existiesen variables con valor fraccionario.

Aquellas soluciones enteras que no requieran la introducción de restricciones dinámicas ni columnas adicionales, actualizarán la cota superior (global) del problema. Por otro lado, el valor objetivo de la última optimización en el nodo dará lugar a la cota inferior del mismo. Una vez actualizada la cota superior, se eliminarán aquellos nodos cuya cota inferior supere o iguale la cota superior global.

Los ficheros que se adjuntan (Véase Apéndice A.9 y A.10) ilustran cómo construir un generador de columnas a partir de la biblioteca BCP que se ha creado.

En primer lugar, se crea un objeto derivado de la clase *pricer_t*. Este objeto tiene un procedimiento virtual, llamado *generate_columns*, que debe ser implementado en la clase hija.

Dentro del constructor se establecen los valores de aquellos atributos específicos del generador de columnas y, dentro del método *generate_columns*, se describe la llamada a los algoritmos de generación dinámica de columnas (que en este caso se llama *SP*). Una vez generadas, se introducirán en el pool y se tomarán en cuenta en el programa principal.

Capítulo III. Estudio Computacional

3. Estudio Computacional

A continuación, se hará un estudio computacional del proyecto que se quiere realizar. Dado que se ha desarrollado una biblioteca de clases, se ha tenido que implementar un problema específico para poder realizar este estudio.

3.1 Descripción del modelo

Esta sección ofrece una descripción informal del problema abordado para el estudio. Se trata del *Problema de Buses de Escuela*, en el que dado un conjunto de localizaciones conectadas entre sí por carreteras – que actuarán como paradas potenciales de bus–, hay un conjunto de usuarios deseando llegar a un destino común; dichos usuarios pueden llegar a alguna(s) de las paradas potenciales a pie. Se posee un depósito central que opera una flota de buses con la misma capacidad, cuyo propósito es llevar a los usuarios al destino. El problema considera dos tipos de coste: un coste de ruta, relacionado con la longitud del camino que conecta dos paradas potenciales, y un coste de asignación, asociado a la distancia caminada por cada usuario a cada parada potencial. Este problema tiene como objetivo asignar a cada usuario una parada potencial y diseñar un conjunto de rutas – que lleguen al destino – que visite todas las paradas con usuarios asignados, de tal forma que el número total de usuarios asignado a las paradas que visita cada ruta no exceda la capacidad del vehículo.

La suma de la longitud de las rutas y de los costes de asignación debe minimizarse. Hay que resaltar que no todas las paradas deben ser visitadas, pero todos los usuarios sí deben ser recogidos por los vehículos.

3.1.1 Ejemplo gráfico del modelo

En la Figura 1, se puede apreciar un ejemplo gráfico para una instancia específica del problema con diferentes valores para la restricción de recursos.

Esta imagen muestra tres soluciones óptimas para una instancia con capacidad 10, donde los puntos negros representan a los usuarios y los grises a las paradas potenciales. La asignación de usuarios a paradas potenciales se representa por la línea de puntos, mientras que cada línea solida indica una ruta de bus entre dos paradas. La Figura 1 (a) muestra una solución donde la cota superior entre rutas viene dada por $L=180$, y la cota inferior para la capacidad de los vehículos no está definida; esta solución recoge 25 usuarios con cuatro rutas. La Figura 1 (b) muestra la solución óptima para la instancia con $L=124$, sin cota inferior definida; esta solución usa siete rutas, pero una de ellas recoge solo un usuario. La Figura 1 (c) añade al ejemplo (b) una cota inferior para la capacidad de los vehículos de $\underline{Q} = 4$; este último utiliza seis rutas, en la que cada una recoge al menos a 4 usuarios.

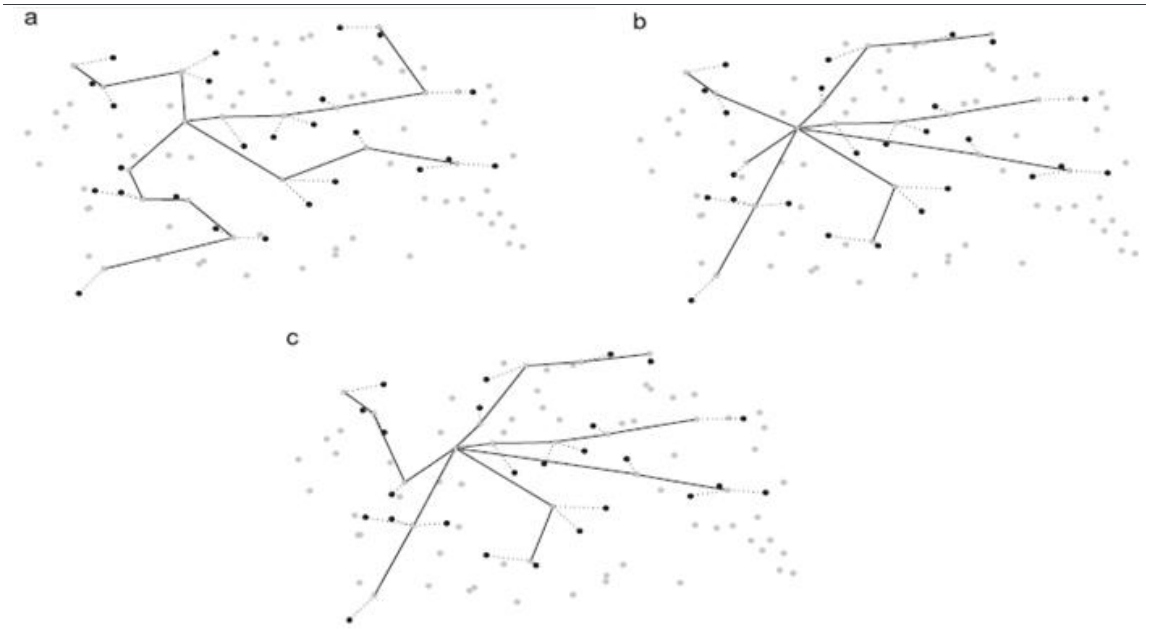


Figura 2 Tres soluciones óptimas para una instancia con capacidad 10: (a) $L = 180,\ \underline{Q} = 0$, (b) $L = 124.\ O = 0$. (c) $L = 124.\ O = 4$.

3.2 Formulaciones del modelo

Ya se ha hecho una descripción del modelo y ahora se procederá a introducir dos formulaciones diferentes del modelo.

3.2.1 Notación matemática

Sea U el conjunto de usuarios y N el conjunto de paradas potenciales. Una flota homogénea de vehículos, denotada por el conjunto K, está disponible en el depósito. La capacidad de cada vehículo viene dada por Q. Por simplicidad, se asume que el depósito y el destino se encuentran en la misma localización, representada por 0.

Para un conjunto dado $W \subseteq U$, $N(W)$ representa el conjunto de paradas potenciales alcanzables por el usuario W. De forma opuesta, para un conjunto dado, $S \subseteq N, U(S)$ denota el conjunto de usuarios que es capaz de alcanzar una parada en S.

Se modela el problema con un grafo directo $G = (V, A)$, donde $V = N \cup \{0\}$, es decir, V es el conjunto de nodos que incluye las paradas potenciales y el depósito. El conjunto de arcos del grafo G, se define como $A = \{(i,j): i \in V, j \in V \backslash \{i\}\}$.

La variable c_{ij} representa el coste de ruta para cada arco $(i,j) \in A$. Para cada par (u,i) que denota una asignación potencial de un usuario u a una parada i, $\forall u \in U, i \in N(u)$, c_{ui} nos dice el coste de asignar u a i.

Se define $\delta^+(S) = \{(i,j) \in A: i \in S, j \notin S\}$ y $\delta^-(S) = \{(i,j) \in A: i \notin S, j \in S\}$ para cualquier subconjunto de nodos $S \subseteq V$. Se denota $A(S,T),\ \forall S, T \subseteq V$ y $S \cap T = \emptyset$, al conjunto de arcos que van de cada nodo de S a cada nodo de T.

3.2.2 Primera formulación

Esta formulación se presenta en base a tres familias de variables discretas. Sea x_{ij}^k, para cada arco $(i,j) \in A$ y para cada vehículo $k \in K$, una variable binaria que tome el valor 1 si el vehículo k incluye en su ruta el arco (i,j) y 0, en otro caso. Sea y_i^k, para cada parada potencial $i \in N$ y $\forall k \in K$, la variable binaria que indica si el vehículo k visita la parada i. Para cada usuario $u \in U$, cada parada potencial $i \in N(u)$, y cada vehículo $k \in K$, se define z_{ui}^k como la variable binaria que toma valor 1 si y solo si el vehículo k recoge al usuario u en la parada i. La siguiente formulación describe el problema:

(21)

$$\sum_{(i,j)\in A} c_{ij} \sum_{k\in K} x_{ij}^k + \sum_{u\in U} \sum_{i\in N(u)} c_{ui} \sum_{k\in K} z_{ui}^k ,$$

(22)

$$\sum_{(i,j)\in \delta^+(i)} x_{ij}^k = y_i^k , \qquad \forall i \in N, \forall k \in K,$$

(23)

$$\sum_{(i,j)\in \delta^-(i)} x_{ij}^k = \sum_{(j,i)\in \delta^-(i)} x_{ji}^k , \qquad \forall i \in V, \forall k \in K,$$

(24)

$$\sum_{(i,j)\in \delta^-(S)} x_{ij}^k \geq y_l^k , \qquad \forall S \subseteq N, l \in S, \forall k \in K,$$

(25)

$$z_{ui}^k \leq y_i^k , \qquad \forall S \subseteq N, \forall i \in N(u), \forall k \in K,$$

(26)

$$\sum_{k\in K} \sum_{i\in N(u)} z_{ui}^k = 1, \qquad \forall u \in U ,$$

(27)

$$\sum_{k\in K} y_i^k \leq 1 , \qquad \forall i \in N,$$

(28)

$$f_r(x^k, y^k, z^k) \leq B_r, \qquad \forall r \in R, \forall k \in K$$

(29)

$$x_{ij}^k \in \{0,1\}, \qquad \forall (i,j) \in A, \forall k \in K,$$

(30)

$$y_i^k \in \{0,1\}, \qquad \forall i \in N, \forall k \in K,$$

(31)

$$z_{ui}^{k} \in \{0,1\}, \qquad \forall u \in U, \forall i \in N(u), \forall k \in K.$$

La función objetivo (21) minimiza la suma de los costes de ruta más los de asignación. La restricción (22) y cota (31) obligan a que exactamente un arco salga de cada parada visitada $i \in N$ y para cada vehículo *k*. La restricción (23), para cada $i \in V$ y cada *k* en *K*, fuerza que salgan el mismo número de arcos que entran. La restricción (24) es la restricción de eliminación de subtour, que asegura que cada parada visitada en un subconjunto $S \subseteq N$, debe conectarse con el depósito a través de un camino. La restricción (25) elimina la posibilidad de asignar a un usuario una parada no visitada. La desigualdad (26) garantiza que cada usuario se recoja una sola vez. La desigualdad (27) obliga a que cada parada sea visitada a lo sumo por un vehículo. La restricción (28) excluye las rutas que supongan un coste mayor a B_r unidades del recurso *r*.

3.2.3 Segunda formulación

Alternativamente, el problema puede modelarse como una formulación de partición de conjunto, derivado de la descomposición de Dantzig-Wolfe [13] sobre la primera formulación. Esta formulación hace una descripción del problema basándose en las variables *x*, *y* y *z*, que representan la agregación de las variables x^k, y^k y z^k para todo vehículo $k \in K$, respectivamente:

(32)

$$\min_{(x,y,z)} \sum_{(i,j)\in A} c_{ij}\mathrm{x_{ij}} + \sum_{u\in U}\sum_{i\in N(u)} c_{ui}\mathrm{z_{ui}},$$

(33)

$$\sum_{i\in N(u)} \mathrm{z_{ui}} = 1, \qquad \forall \mathrm{u} \in \mathrm{U},$$

(34)

$$x_{ij} \in \{0,1\}\ \forall (i,j) \in A, \qquad y_i \in \{0,1\}\ \forall i \in N,$$

(35)

$$z_{ui} \in \{0,1\}\ \forall u \in U, \forall i \in N(u), \qquad x_{ij} = \sum_{p\in\Omega} x_a^p \lambda^p \quad (i,j) \in A,$$

(36)

$$y_i = \sum_{p\in\Omega} y_i^p \lambda^p \quad \forall i \in N,$$

(37)

$$z_{ui} = \sum_{p\in\Omega} z_{ui}^p \lambda^p \quad \forall u \in U, \forall i \in N(u),$$

(38)

$$\lambda^p \geq 0 \quad \forall p \in \Omega.$$

La restricción (33) y cota (34) realizan la misma función que (26) y (27), respectivamente. Las restricciones (33) y (38) garantizan la convexidad de las variables λ en esta formulación. Las igualdades (35), (36) y (37) definen el emparejamiento de las variables *x*, *y* y *z* con λ, respectivamente.

3.2.4 Algoritmo Branch-and-Cut-and-Price

A continuación, se presenta una descripción del algoritmo basado en los modelos previamente descritos. La siguiente figura muestra el procedimiento general de acotación para un algoritmo Branch-and-Cut-and-Price (véase Figura 2)

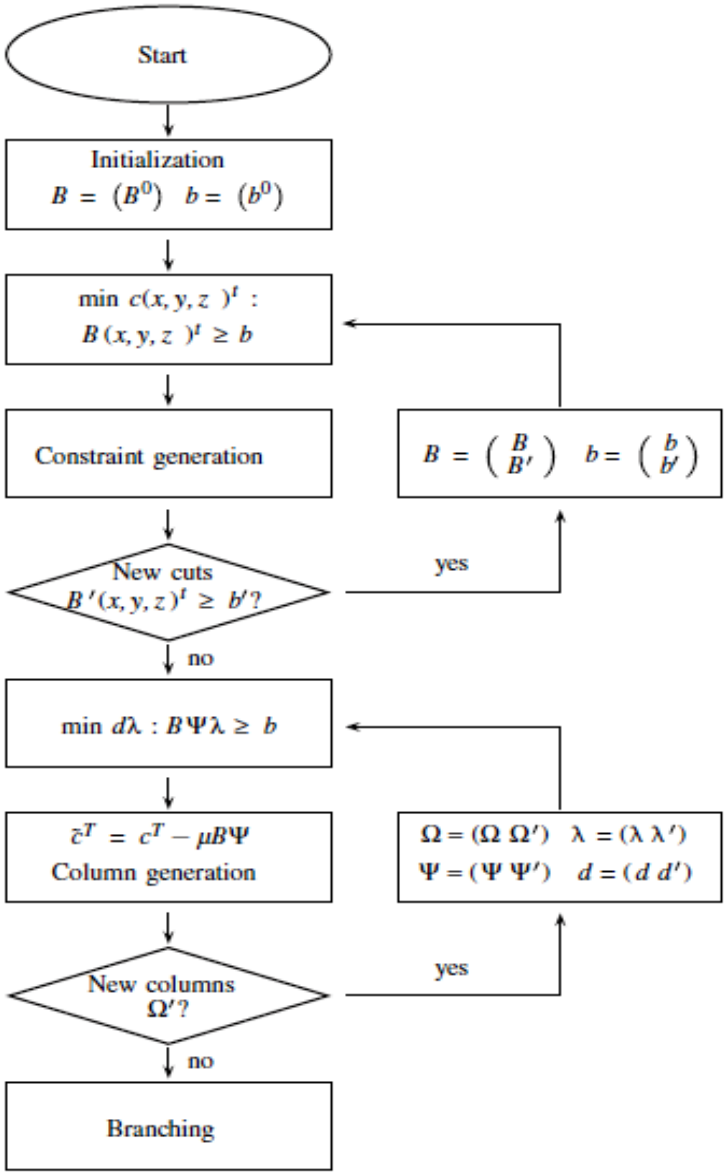


Figura 3 Procedimiento general de acotación para un algoritmo BCP

3.3 Análisis global del modelo

Una vez descrito el problema abordado para el análisis de la biblioteca que se pretende implementar, podemos realizar una comparación de resultados computacionales, donde:

- *Q* = capacidad del vehículo
- *L* = limitación en kilómetros de la ruta
- *(a)*, *(b)* y *(c)*, son los diferentes escenarios
- *%LB* = cota inferior (en porcentaje)
- *Nodes* = número de nodos visitados
- *Cols* = número de columnas añadidas
- *Cuts* = número de cortes realizados
- *CPU* = tiempo de ejecución

Q	L	(a)					(b)					(c)				
		%LB	*nodes*	*cols*	*cuts*	*CPU*	*%LB*	*nodes*	*cols*	*cuts*	*CPU*	*%LB*	*nodes*	*cols*	*cuts*	*CPU*
10	277	95.8	437	396 930	1402	76	96.0	709	704 118	0	80	96.0	377	382 994	653	46
	238	95.6	129	121 228	214	11	97.0	135	110 101	0	16	97.0	202	178 866	276	25
	222	97.5	45	44 005	89	4	98.2	48	36 774	0	6	98.3	19	16 857	9	2
	221	94.3	401	393 123	305	23	95.1	358	298 193	0	28	95.1	797	698 041	508	67
	216	93.5	424	386 797	800	22	94.2	372	299 557	0	28	94.2	382	328 607	204	29
	215	93.4	458	423 226	789	28	94.0	307	239 441	0	23	94.0	267	216 147	110	22
	214	88.4	25 895	2 785 5592	26 561	1336	88.9	23 866	2 258 8777	0	1414	88.9	21 681	2 106 1059	13 078	1438
	210	89.2	10 535	1 054 3897	12 426	466	89.2	13 333	1 185 7040	0	761	89.2	21 581	2 042 0738	9361	1274
	191	95.1	29	18 497	83	1	95.1	13	7457	0	1	95.1	24	14 188	16	1
	190	93.4	49	32 576	67	2	93.4	31	18 420	0	1	93.4	29	18 744	14	2
	187	93.7	48	29 597	72	2	93.7	31	18 344	0	2	93.7	41	25 108	21	2
	186	88.4	3729	3 115 193	1744	91	88.4	3475	2 753 513	0	125	88.4	2719	2 151 782	377	96
	165	100.0	1	394	0	0	100.0	1	521	0	0	100.0	1	521	0	0
	164	95.2	3331	2 277 806	1482	53	95.2	1725	1 077 168	0	37	95.2	1721	1 073 878	7	35
7	222	97.1	225	146 443	448	6	98.4	63	34 600	0	3	98.4	63	34 848	68	2
	218	94.3	1783	1 371 349	2815	49	95.6	521	325 478	0	19	95.6	561	371 279	797	22
	210	94.2	1295	1 051 468	1157	39	95.2	277	172 683	0	10	95.2	209	131 002	118	8
	191	96.5	21	11 456	87	1	96.9	29	14 838	0	1	96.9	29	15 290	49	1
	190	94.8	89	53 102	93	2	95.2	29	14 278	0	1	95.3	27	13 963	33	1
	187	94.5	69	36 904	63	2	94.6	47	21 396	0	2	94.6	97	47 517	84	3
	186	89.2	4822	3 915 765	2275	85	89.3	1881	1 281 254	0	55	89.3	2056	1 390 031	779	61
	165	100.0	4	1514	0	0	100.0	2	927	0	0	100.0	2	927	0	0
	164	95.2	2721	1 876 055	1285	27	95.2	559	344 416	0	9	95.2	677	435 762	219	11
6	222	96.8	853	573 666	1483	14	97.0	481	248 893	0	10	97.0	403	219 772	396	10
	214	96.7	579	379 785	1031	10	96.8	436	228 075	0	10	96.8	423	224 922	598	10
	192	97.0	28	11 164	92	1	97.1	29	10 633	0	1	97.1	45	17 011	42	1
	191	96.0	37	17 150	109	1	96.1	39	15 204	0	1	96.1	31	11 594	28	1
	187	95.9	52	22 816	48	1	95.9	83	36 192	0	2	95.9	39	15 501	42	1
	186	93.1	371	207 219	220	6	93.1	662	355 660	0	14	93.1	283	152 282	28	7
	165	100.0	1	400	0	0	100.0	4	1720	0	0	100.0	4	1720	0	0
	164	95.2	1693	1 083 875	1543	16	95.2	2584	1 938 658	0	34	95.2	2392	1 617 435	1055	32
		94.8	1940.5	1 819 322.3	2089.8	76.6	95.2	1681.6	1 453 365.5	0.0	86.9	95.2	1844.6	1 654 464.1	934.5	103.6

Tabla 1 Cómputo de resultados para diferentes escenarios

3.4 Análisis para una instancia del modelo

Como ya se ha hecho un análisis global del problema de Buses de Escuela, ahora podemos hacer un análisis específico, que consiste en una tabla que muestra los resultados de una ejecución para una entrada del problema. Las variables a considerar son:

- *CPU* = tiempo de ejecución
- *Node* = identificador del nodo en ejecución
- *Left* = número de nodos restantes
- *LP it* = número de iteraciones en el nodo en cuestión
- *Mem* = cantidad de memoria ocupada
- *Frac* = número de variables con resultado fraccionario
- *P. Cols* = número de columnas generadas hasta el momento
- *Cols* = número de columnas generadas en ese nodo
- *Cuts* = número de cortes generados en ese nodo
- *Cons* = número de restricciones del modelo
- *Vars* = número de variables del modelo
- *Dual bound* = cota inferior global
- *Primal bound* = cota superior global
- *Gap* = ratio entre cotas inferior y superior.

Así, se pueden ver los resultados para las dos últimas iteraciones (Véase Tabla 2 y Tabla 3)

CPU	node	left	LP it	Mem	frac	P. cols	cols	cuts	cons	vars	dual bound	primal bound	gap
1.0s	743	60	2	6096K	0	1640	20	1	15	56	1,38E+11	1,40E+11	1.23%
1.0s	504	59	2	6096K	0	1642	18	1	14	56	1,38E+11	1,40E+11	1.23%
1.0s	505	58	1	6096K	3	1644	18	0	13	56	1,38E+11	1,40E+11	1.23%
1.0s	510	59	2	6096K	0	1646	12	1	11	56	1,38E+11	1,40E+11	1.23%
1.0s	511	58	1	6096K	3	1648	12	0	10	56	1,38E+11	1,40E+11	1.23%
1.0s	514	59	2	6096K	0	1650	12	1	11	56	1,38E+11	1,40E+11	1.23%
1.0s	515	58	1	6096K	6	1652	12	0	10	56	1,38E+11	1,40E+11	1.23%
1.0s	544	59	1	6096K	3	1654	16	0	12	56	1,38E+11	1,40E+11	1.23%
1.0s	545	60	2	6096K	0	1656	16	1	13	56	1,38E+11	1,40E+11	1.23%
1.0s	558	59	1	6096K	3	1658	18	0	13	56	1,38E+11	1,40E+11	1.23%
1.0s	559	60	1	6096K	6	1660	18	0	13	56	1,38E+11	1,40E+11	1.23%
1.0s	584	61	2	6096K	0	1662	24	1	17	56	1,38E+11	1,40E+11	1.23%
1.0s	585	60	2	6096K	0	1664	24	1	17	56	1,38E+11	1,40E+11	1.23%
1.0s	658	59	1	6096K	7	1666	28	0	20	56	1,38E+11	1,40E+11	1.23%
1.0s	659	60	1	6096K	10	1668	28	0	20	56	1,38E+11	1,40E+11	1.23%
1.0s	862	61	2	6132K	0	1670	34	1	23	56	1,38E+11	1,40E+11	1.23%
1.0s	863	60	2	6096K	0	1672	34	1	23	56	1,38E+11	1,40E+11	1.23%
1.1s	882	59	2	6096K	0	1674	20	1	15	56	1,38E+11	1,40E+11	1.23%
1.1s	883	58	2	6096K	0	1676	20	1	15	56	1,38E+11	1,40E+11	1.23%
1.1s	884	57	2	6096K	0	1678	14	1	12	56	1,38E+11	1,40E+11	1.23%

Tabla 1 Tabla para una entrada del problema (penúltima ejecución)

CPU	node	left	LP it	Mem	frac	P. cols	cols	cuts	cons	vars	dual bound	primal bound	gap
1.1s	885	56	2	6096K	0	1680	14	1	12	56	1,38E+11	1,40E+11	1.23%
1.1s	886	55	1	6096K	3	1682	14	0	11	56	1,38E+11	1,40E+11	1.23%
1.1s	887	56	1	6096K	8	1684	14	0	11	56	1,38E+11	1,40E+11	1.23%
1.1s	888	57	2	6096K	0	1686	18	1	14	56	1,38E+11	1,40E+11	1.23%
1.1s	889	56	2	6096K	0	1688	18	1	14	56	1,38E+11	1,40E+11	1.23%
1.1s	890	55	1	6096K	3	1690	20	0	14	56	1,38E+11	1,40E+11	1.23%
1.1s	891	56	2	6096K	0	1692	20	1	15	56	1,38E+11	1,40E+11	1.23%
1.1s	892	55	2	6100K	0	1694	20	1	15	56	1,38E+11	1,40E+11	1.23%
1.1s	893	54	2	6100K	0	1696	20	1	15	56	1,38E+11	1,40E+11	1.23%
1.1s	894	53	1	6100K	0	1698	30	0	21	56	1,38E+11	1,38E+11	0.00%

Tabla 2 Tabla para una entrada del problema (última ejecución)

Capítulo IV. Conclusiones y Trabajos Futuros

4. Conclusiones y Trabajos Futuros

4.1 Resumen

Durante el desarrollo de este proyecto hemos estudiado las técnicas de ramificación y poda para la resolución de Programas Lineales Enteros Mixtos, con el fin de desarrollar un marco de trabajo, concretado en una biblioteca de clases. Se pretende que esta biblioteca facilite la implementación de aplicaciones para resolver problemas de optimización.

El desarrollo de esta biblioteca incluye clases para gestionar el árbol de ramificación asociado a la técnica de ramificación y poda. Además, hemos incluido dentro de cada uno de los nodos del árbol la posibilidad de enriquecer el modelo dinámicamente con filas y columnas. Así, el desarrollo final de trabajo ha sido un marco para resolver problemas mediante la técnica Branch-and-Cut-and-Price.

Esta biblioteca ha sido probada sobre dos problemas de optimización en rutas, mostrando un comportamiento robusto y eficiente. La experiencia computacional nos ha mostrado también que este entorno es capaz de resolver problemas de cardinalidad realista.

Los cocimientos necesarios para este proyecto han sido obtenidos de las asignaturas *Algoritmos y Estructuras de Datos*, *Algoritmos y Estructuras de Datos Avanzadas*, *Optimización*, *Análisis y Diseños de Algoritmos* y *Complejidad Computacional*.

4.2 Trabajos Futuros

El trabajo que hemos llevado a cabo durante los plazos previstos para su desarrollo han alcanzado las expectativas. El objetivo ha sido el desarrollo de un prototipo que permita resolver problemas generales descritos mediante Programación Lineal Entera, y que requieran la generación de filas y columnas.

El sentido de este proyecto no era tanto el desarrollo de un producto final, sino de un prototipo que fuera una herramienta pedagógica para la comprensión de algoritmos como Branch-and-Cut-and-Price en el ámbito académico. Sin duda, la brevedad del tiempo de ejecución de trabajo ha dejado flecos por resolver, aunque las directrices ya están claramente establecidas.

Entre estos flecos, hemos observados que se debe mejorar la gestión de filas y columnas, de manera que se puedan eliminar aquellas que no sean necesarias en determinado punto de la ejecución del algoritmo. En otras palabras, se debe mejorar la aplicación de manera que se pueda detectar aquellas restricciones que no intervengan en la resolución del programa lineal y aquellas columnas con costo reducido positivo.

Por otro lado, un futuro proyecto debería incluir la generación de soluciones factibles, es decir, lograr el cálculo de cotas superiores de forma general, ya que hasta el momento sólo se están tomando como cotas superiores aquellas soluciones factibles que se encuentran durante el desarrollo del árbol de ramificación.

Finalmente, observamos que nuestra propuesta sólo puede utilizar un resolutor lineal comercial (CPLEX). Sería una buena práctica modificar la biblioteca de manera que la resolución del programa lineal fuera independiente del resolutor.

Capítulo V. Summary and Conclusions

5. Summary and Conclusions

5.1 Summary and Conclusions

Throughout this project, we have studied variations of the Branch and Bound technique, used for solving mixed integer linear programming problems, with the purpose of creating a development framework. To be more specific, we are talking about a library which allows the implementation of computer applications for solving optimization problems.

The library mentioned above, provides with data structures to manage the branching tree related to the Branch-and-Bound technique. Moreover, we have included the possibility of enriching the model by adding rows and columns dynamically. Therefore, we have finally developed a framework for solving optimization problems by means of the Branch-and-Cut-and-Price technique.

We have tested this work on two optimization routing problems, proving a robust and efficient performance. The computational experience has also proven that this framework is able to solve problems with a realistic cardinality.

The background required for this project has been drawn from the modules *Algorithms and Data Structures*, *Advanced Algorithms and Data Structures*, *Optimization*, *Algorithms Analysis and Design* and *Computational Complexity*.

5.2 Future Work

The work we have been carrying out for the whole semester to develop the project has reached our expectations. The aim has been the development of a template for solving general problems able to be defined as Integer Linear Programming Problems, in addition to both, a row and column generation requirement.

The main goal of this project was not the development of a final product, but a prototype which could be used as a pedagogical tool to help us understanding Branch-and-Cut-and-Price algorithms in academia. There is no doubt that, because of the tight time window in our planning, this work has left some loose ends to solve out; however, the guideline has been clearly established.

Among the possible improvements, we have observed that row and column management should be upgraded, in a way that rows and columns are eliminated when they are no longer needed in some point of the execution of the algorithm. In other words, the application should be improved so that constraints which do not interfere with the linear program solving, are detected, so the columns with positive reduced cost.

On the other hand, a future work of the project could entail feasible solution generation, it means, accomplishing the upper bound calculation in a general form, since feasible solutions found during the execution of the branching tree are the only ones that have been taken into account so far. Finally, it is also shown that our approach is just able to use a commercial-aimed linear solver (CPLEX). It might be a good practice to modify the library and make the linear program solving an independent module, out of the solver scope.

Apéndice A. Códigos

A. 1 Variables (Código fuente)

```
// variables.cpp

const int n_x = n_arc * m ;              // Numero de variables x
const int n_e_k = n_v * m;               // Numero de variables e
const int n_s   = n_v;                    // Numero de variables s
const int n_var = n_x + n_e_k + n_v;     // Numero total de variables

varDes_.init(n_var);

const int base_x = 0;
const int base_e = n_x;
const int base_s = base_e + n_e_k;

for (int i = base_x; i < n_x; i++){
    varDes_.vt_[i] = BCP_template::_CONTINUOUS_;
    varDes_.st_[i] = BCP_template::_SUBPROBLEM_;
    varDes_.c_[i]  = 0.0;
    varDes_.lb_[i] = 0.0;
    varDes_.ub_[i] = 1.0;
}

for (int i = base_e; i < base_e + n_e_k; i++){
    varDes_.vt_[i] = BCP_template::_CONTINUOUS_;
    varDes_.st_[i] = BCP_template::_SUBPROBLEM_;
    varDes_.c_[i] = 0.0;
    varDes_.lb_[i] = 0.0;
    varDes_.ub_[i] = 1E100;
}

for (int i = base_s; i < base_s + n_s; i++){
    varDes_.vt_[i] = BCP_template::_CONTINUOUS_;
    varDes_.st_[i] = BCP_template::_MASTER_PROBLEM_;
    varDes_.c_[i] = 0.0;
    varDes_.lb_[i] = 0.0;
    varDes_.ub_[i] = 1E100;
}
```

A. 2 Restricciones (Código fuente)

```
// restricciones.cpp

for (int j = 0; j < n_v; j++) {
    BCP_template::value_t*     values = new BCP_template::value_t [1];
    BCP_template::long_size_t* columns = new BCP_template::long_size_t [1];

    columns[0] = base_s + j;
    values[0] = 1.0;

    BCP_template::rowData_t* row = new BCP_template::rowData_t(1, columns, values,
-1E100, 0, n_row_);
    n_row_++;

    rowPool_.addRow(row);
}

////////////////////////////////////////////////////////////////////////////
for (int k = 0; k < m; k++)
```

```
    for (int j = 0; j < n_v; j++) {
        BCP_template::value_t*     values = new BCP_template::value_t [2];
        BCP_template::long_size_t* columns = new BCP_template::long_size_t [2];

        columns[0] = base_e + k * n_v + j;
        values[0] = 1.0;

        columns[1] = base_s + j;
        values[1] = -1.0;

        BCP_template::rowData_t* row = new BCP_template::rowData_t(2, columns,
values, -1E100, 0, n_row_);
        n_row_++;

        rowPool_.addRow(row);
    }

//////////////////////////////////////////////////////////////////////////////

for (int j = 1; j < n_v; j ++) {
    BCP_template::value_t*     values = new BCP_template::value_t [(n_v - 1) * m];
    BCP_template::long_size_t* columns = new BCP_template::long_size_t [(n_v - 1)
* m];

    int ll = 0;

    for (int k = 0; k < m; k ++)
        for (int i = 0; i < n_v; i++)
            if (i != j) {
                columns[ll] = k * n_arc + i * n_v + j;
                values[ll] = 1.0;
                ll++;
            }

    BCP_template::rowData_t* row = new BCP_template::rowData_t(ll, columns,
values, I_.q_[j], I_.q_[j], n_row_);

    n_row_++;

    rowPool_.addRow(row);
}

//////////////////////////////////////////////////////////////////////////////

for (int k = 0; k < m; k++) {

        BCP_template::value_t*     values = new BCP_template::value_t [n_v];
        BCP_template::long_size_t* columns = new BCP_template::long_size_t [n_v];

        int ll = 0;

        for (int i = 1; i < n_v; i++) {
            columns[ll] = k * n_arc + 0 * n_v + i;
            values[ll] =  1.0;
            ll++;
        }

        BCP_template::rowData_t* row = new BCP_template::rowData_t(ll, columns,
values, 1, 1, n_row_);
        n_row_++;

        rowPool_.addRow(row);
}

//////////////////////////////////////////////////////////////////////////////
for (int k = 0; k < m; k++)
```

```
    for (int j = 1; j < n_v; j++) {

        BCP_template::value_t*     values  = new BCP_template::value_t [n_v];
        BCP_template::long_size_t* columns = new BCP_template::long_size_t [n_v];

        int ll = 0;

        for (int i = 0; i < n_v; i++)
            if (i != j) {
                columns[ll] = k * n_arc + j * n_v + i;
                values[ll] =  1.0;
                ll++;
            }

        BCP_template::rowData_t* row = new BCP_template::rowData_t(ll, columns,
values, -1E100, 1, n_row_);
        n_row_++;

        rowPool_.addRow(row);
    }
```

A. 3 Árbol de ramificación (Archivo cabecera)

```
// BCP_branching_tree_t.hpp

class branching_tree_t {
public:
    statistics_t& statistics_;

    sll tree_;
    sll already_processed_;

public:
    branching_tree_t(statistics_t& statistics, branching_node_t* node);

    virtual ~branching_tree_t(void);
    virtual void process_tree(void);
};
```

A. 4 Árbol de ramificación (Código fuente)

```
// BCP_branching_tree_t.cpp

branching_tree_t::branching_tree_t(statistics_t& statistics, branching_node_t*
node) :
tree_(),
node_counter_(1) {
    push(node);
}

branching_tree_t::~branching_tree_t(void) {
    tree_.clean();
    already_processed_.clean();
}

void branching_tree_t::process_tree(void) {¡
    while (!tree_.empty()) {
        branching_node_t* current_node = pop();

        current_node->activate();
        current_node->optimize();
```

```
        if (current_node->dualSolution_->lB_ + integer_tolerance_ > current_node-
>solution_.uB_) {
            current_node->status_ = PRUNED;

            do_branch(current_node);

            store_node(current_node);
            current_node = NULL;
        }
    }
    tree_.clean();
    already_processed_.clean();
}
```

A. 5 Nodo (Archivo cabecera)

```
// BCP_branching_node_t.hpp

class branching_node_t {
public:
    statistics_t& statistics_;
    problemInfo_t& problemInfo_;
    problemSolution_t& solution_;

    pricer_t& pricer_;
    separator_t& separator_;

    primal_heuristic_t& heuristic_;

    basic_colDataProblemPool_t& problemCols_; // Todas las columnas generadas
    rowDataProblemPool_t& problemRows_;       // Todas las filas generadas

    basic_colDataNodePool_t nodeCols_;
    // Todas las columnas heredadas y las generadas
    rowDataNodePool_t nodeRows_;
    // Todas las filas heredadas y las generadas

    dualSolution_t*   dualSolution_;          // Solución en cada iteración
    nodeSolver_t*     nodeSolver_;            // Optimizador del nodo

    branching_node_t** children_;
    int               n_children_;

public:
    branching_node_t(master_problem_t*, obj_inx_t node_id, double slack_var_cost =
SLACK_VAR_COST);
    virtual void optimize(void);
};
```

A. 6 Nodo (Código fuente)

```
// BCP_branching_node_t.cpp

branching_node_t::branching_node_t(master_problem_t* mp, obj_inx_t node_id, double
slack_var_cost) :
statistics_(mp->statistics_),
problemInfo_(mp->problemInfo_),
solution_(mp->solution_),
pricer_(mp->pricer_),
separator_(mp->separator_),
heuristic_(mp->heuristic_),
```

```
problemCols_(mp->problemCols_),
problemRows_(mp->problemRows_),
nodeCols_(mp->problemInfo_.n_problem_var_),
nodeRows_(),
dualSolution_(NULL),
nodeSolver_(NULL),
children_(NULL),
n_children_(0),
nodelB_(-BD_INFINITY)
{}

void branching_node_t::optimize(void) {
    do {
        const int nRow = nodeRows_.nRow();
        const int nCol = nodeCols_.nCol();
        new_cols = new_rows = 0;

        //------------------------------------------------------------------
        // Solving
        //------------------------------------------------------------------
        nodeSolver_->solve(nRow, nCol, dualSolution_->lB_, dualSolution_->x_s_,
dualSolution_->x_, lambda, rc, dualSolution_->pi_, dualSolution_->slack_,
dualSolution_->lpFeasible_);

        if (dualSolution_->lpFeasible_) {

            //------------------------------------------------------------------
            // Pricing
            //------------------------------------------------------------------
            if (new_rows == 0) {
                pricer_.pricing(nodeRows_, colPool, nodeInfo_->fixed_to_zero_,
nodeInfo_->fixed_to_one_, dualSolution_->pi_);
                dualSolution_->n_found_col_ = colPool.nCol();

                new_cols = colPool.nCol();
                problemCols_.moveFrom(colPool);
            }

            //------------------------------------------------------------------
            // Separating
            //------------------------------------------------------------------
            if (new_cols == 0) {
                separator_.separate(dualSolution_->x_, rowPool);

                new_rows = rowPool.nRow();
                problemRows_.moveFrom(rowPool);
            }
        }
    } while (new_rows + new_cols != 0);
}
```

A. 7 Separador (Archivo cabecera)

```
// BCP_separator_t.hpp

class benders_cut_generator_t: public BCP_template::separator_t {
public:
    benders_cut_generator_t(IAV_Instance& I);
    virtual ~benders_cut_generator_t(void);
    virtual void separate(double* x, double lb, BCP_template::nodeInfo_t* ni
,BCP_template::rowDataPreliminarPool_t& pool);
```

```
    virtual bool feasible(double* x, double lb, BCP_template::nodeInfo_t* ni );
};
```

A. 8 Separador (Código fuente)

```
// BCP_separator_t.cpp

benders_cut_generator_t::benders_cut_generator_t(IAV_Instance& I):
BCP_template::separator_t()
{}

benders_cut_generator_t::~benders_cut_generator_t(void)
{}

void benders_cut_generator_t::separate(double* x, double lb,
BCP_template::nodeInfo_t* ni ,BCP_template::rowDataPreliminarPool_t& pool) {
    // Encontrar una desigualdad violada
}

bool benders_cut_generator_t::feasible(double* x, double lb,
BCP_template::nodeInfo_t* ni) {
    return true;
}
```

A. 9 Pricer (Archivo cabecera)

```
// BCP_pricer_t.hpp

class iav_pricer_t  : public BCP_template::pricer_t {
public:
    IAV_Instance& I_;
    double Cap_;
    DP_2_cyc_CAP_shortest_path_CMT SP_;

public:
    iav_pricer_t(int nVar, double* c, IAV_Instance& I);
    iav_pricer_t(IAV_Instance& I);
    virtual ~iav_pricer_t(void);

    virtual void generateColumns(int nvar, double* rc,int nrow, double* pi,
bit_set& fixed_to_zero, bit_set& fixed_to_one,
BCP_template::basic_colDataPreliminarPool_t& p_coolPool);
};
```

A. 10 Pricer (Código fuente)

```
// BCP_pricer_t.cpp

iav_pricer_t::iav_pricer_t(int nVar, double* c, IAV_Instance& I):
pricer_t(nVar,c),
I_(I),
SP_(I.n_, I.m_, 100000.0, I.r_, I.m_t_, I.t_)
{}

iav_pricer_t::iav_pricer_t(IAV_Instance& I):
pricer_t(),
I_(I),
SP_(I.n_, I.m_, 100000.0, I.r_, I.m_t_, I.t_)
{}

iav_pricer_t::~iav_pricer_t(void)
{}

void iav_pricer_t::generateColumns(int nvar, double* rc, int nrow, double* pi,
bit_set& fixed_to_zero, bit_set& fixed_to_one,
BCP_template::basic_colDataPreliminarPool_t& p_colPool) {
    const int m         = I_.m_;
    const int n_v       = I_.n_ + 1;
    const int n_arc     = n_v * n_v;

    double sg[nvar];

    for(int i = 0; i < nvar; i++)
        if ((fixed_to_zero.contains(i)))
            sg[i] = MAX_DP_COST;
        else
            sg[i] = rc[i];

    double cost[n_arc + n_v];

    for(int k = 0; k < m; k++) {
        std::vector<path_t*> path_list;

        update_red_cost(sg, k, cost);
        SP_.solve(cost, k, path_list);

        const int pl_sz = path_list.size();
        path_t* aux;

        for(int i=0;i<pl_sz;i++) {
```

```
            aux = path_list[i];

            if (aux->redcost_< -1E-2) {
                addCol(p_colPool, aux->nit_, aux->inx_, aux->val_);
                delete aux;
            }
            else {
                aux->clean();
                delete aux;
            }
        }
    }
}
```

Bibliografía

[1] «Computational Infrastructure for Operations Research,» COIN-OR Foundation, 22 Enero 2012. [En línea]. Available: http://www.coin-or.org/. [Último acceso: 25 Abril 2014].

[2] T. Achterberg, T. Berthold, T. Koch y K. Wolter, «Solving Constraint Integer Programs,» 23 Marzo 2014. [En línea]. Available: http://scip.zib.de/. [Último acceso: 25 Abril 2014].

[3] R. Sala Garrido, Programación Lineal: Metodología y Problemas, Tebar, 1993.

[4] D. de la Fuente García y P. Priore Moreno, «Programación Lineal Entera y Programación no Lineal,» Oviedo, España, Servicio de Publicaciones de la Universidad de Oviedo, 1996, pp. 1-13.

[5] R. Neapolitan y K. Naimipour, «Foundations of Algorithms Using C++ Pseudocode (3rd Edition),» pp. 233-264.

[6] V. Quimey, «Capitulo 4: Programación Lineal Entera,» 19 Diciembre 2013. [En línea]. Available: http://mate.dm.uba.ar/~qvivas/operativa/archivos/cap4_05.pdf. [Último acceso: 3 Mayo 2014].

[7] A. Caprara y M. Fischetti, «Branch and Cut Algorithms,» Annotated bibliographies in combinatorial optimization, 1997, p. Chapter 4.

[8] M. Jünger y S. Thienel, «Introduction to ABACUS – a Branch-and-Cut System,» Operations Research Letters, 1992, p. 22.

[9] M. Padberg y G. Rinaldi, «A branch and cut algorithm for the resolution of large-scale symmetric traveling salesman problems,» SIAM Review,, 1991, p. 33.

[10] S. Thienel, «ABACUS: A Branch and Cut System,» Universität zu Köln, 1995.

[11] C. Barnhart, E. Johnson, G. Nemhauser, M. Savelbergh y P. Vance, «Branch-and-Price: Column Generation for Solving Huge Integer Programs,» Braun-Brumfield, 1994, pp. 186-207.

[12] F. Vanderbeck y L. Wolsey, Reformulation and Decomposition of Integer Programs, Springer, Berlin: 50 Years of Integer Programming, 2010.

[13] G.-. G.-. Dantzig y P. Wolfe, Decomposition principle for linear programs, Oper. Res.,

1960.

[14] J. J. Salazar-González, Programación Matemática, Ediciones Días de Santos, 2001.

[15] E. Balas, S. Ceria, G. Cornuéjols y N. Natraj, «Gomory Cuts Revisited,» Operations Research Letters, 1996, p. 19.

[16] P. Toth y D. Vigo, The Vehicle Routing Problem, S.I.A.M., 2002.

[17] T. B. T. K. K. W. Tobias Achterberg, «Solving Constraint Integer Programs,» 23 Marzo 2014. [En línea]. Available: http://scip.zib.de/. [Último acceso: 25 Abril 2014].

[18] R. S. Garrido, Programación Lineal: Metodología y Problemas, Tebar, 1993.

[19] P. P. M. David de la Fuente García, «Programación Lineal Entera y Programación no Lineal,» Oviedo, España, Servicio de Publicaciones de la Universidad de Oviedo, 1996, pp. 1-13.

[20] Q. Vivas, «Capitulo 4: Programación Lineal Entera,» 19 Diciembre 2013. [En línea]. Available: http://mate.dm.uba.ar/~qvivas/operativa/archivos/cap4_05.pdf. [Último acceso: 3 Mayo 2014].

[21] K. N. Richard E. Neapolitan, «Foundations of Algorithms Using C++ Pseudocode (3rd Edition),» pp. 233-264.

[22] M. F. A. Caprara, «Branch and Cut Algorithms,» Annotated bibliographies in combinatorial optimization, 1997, p. Chapter 4.