Universidad de La Laguna

Sección de Matemáticas | Facultad de Ciencias

Samuel Santos Lucas Castilla

Problema del Viajante de Comercio con recogida y entrega de mercancía

Pick-up and Delivery Travelling Salesman Problem (1-PDTSP)

Trabajo Fin de Grado
Grado en Matemáticas
La Laguna, Julio de 2018

Dirigido por

Hipólito Hernández Pérez

Hipólito Hernández Pérez
Departamento de Matemáticas, Estadística e Investigación Operativa
Universidad de La Laguna
38271 La Laguna, Tenerife

Agradecimientos

A mi tutor Hipólito Hernández Pérez
por haber aceptado dirigir este trabajo
y por haber aportado tanta
dedicación y guía para su desarrollo.
A toda mi familia y amigos,
en especial a mis padres y mi hermana
, por su apoyo incondicional.

Resumen · Abstract

Resumen

Este Trabajo Fin de Grado ha afrontado una variante de uno de los problemas más importantes, y por ello más estudiado, de la optimización combinatoria, el problema del viajante de comercio (TSP), en el cual partiendo de un depósito se debe visitar un número de localizaciones de forma que una vez realizado el recorrido se regresa al depósito y de tal manera que se minimice el costo de dicho recorrido. Concretamente abordamos el problema llamado Problema del viajante de comercio con recogida y entrega de mercancía (1-PDTSP). El 1-PDTSP tiene como objetivo el mismo que el TSP, pero con el añadido de que cada localización debe recibir o proporcionar una cierta cantidad de producto que es transportada por un único vehículo, el cual posee una capacidad máxima de carga.
El objetivo de este trabajo es identificar, estudiar y conocer los métodos y modelos matemáticos mediante los cuales se resuelva de manera efectiva nuestro problema.
Finalmente, se puede considerar este trabajo como una extensión de los conocimientos que se han adquirido en la asignatura de Programación Combinatoria del Grado de Matemáticas de la Universidad de la Laguna, puesto que con el desarrollo del trabajo se ha reflexionado e investigado más sobre los métodos de resolución de los problemas lineales enteros, como por ejemplo el método de ramificación y corte, y sobre la posterior modelización y programación informática de los mismos.

Palabras clave: *TSP – 1-PDTSP – Modelos matemáticos – Optimización Combinatoria.*

Abstract

This memory has been tackled a variant of the most important and therefore most studied problem of combinatorial optimization, the Traveling Saleman Problem, in which starting from a deposit, a number of locations must be visited so that once the tour has been completed, it is returned to the deposit, and in such a way as to minimize the cost of the tour. More specifically, we address the problem called the Pick-up and Delivery Traveling Salesman Problem. This problem has the same objective as the TSP, but with the addition that each location must receive or provide a certain amount of product that is transported by a single vehicle, which has a maximum load capacity. The aim of this paper is to identify, study and understand the mathematical methods and models, belonging to the field of mathematical programming and specifically to combinatorial optimization, through which our problem is effectively resolved.

Finally, this work can be considered an extension of the knowledge acquired in the subject Combinatorial Optimization of the Mathematics Degree at the University of La Laguna. The develomente of this work has inproved my knowledge of combinatorial optimization (i.e., integer programming, branching and cutting methods) and computer programing.

Keywords: *Traveling Saleman Problem – Pick-up and Delivery – Mathematical methods and models – Combinatorial optimization.*

Contenido

Agradecimientos III

Resumen/Abstract V

Introducción IX

1. Fundamentos teóricos y notación 1
- 1.1. Programación Matemática 1
- 1.2. Problemas de rutas 3
  - 1.2.1. Problema del Viajante de Comercio 3
  - 1.2.2. Problemas de rutas de vehículos 6
- 1.3. Método de resolución: Algoritmo de ramificación y corte 9
- 1.4. Material y herramientas de trabajo 10

2. Problema del Viajante de Comercio con recogida y entrega de mercancía 13
- 2.1. Introducción 13
- 2.2. Modelos matemáticos para el 1-PDTSP 14
  - 2.2.1. Modelo con restricciones de subciclos 14
  - 2.2.2. Modelo 1-PDTSP carga-flujo 15
  - 2.2.3. Modelo 1-PDTSP multiflujo 16
  - 2.2.4. Modelo 1-PDTSP con variable potencial 17
  - 2.2.5. Modelo 1-PDTSP con restricciones de capacidad 18

3. Implementación en Xpress-Mosel y resultados computacionales 21
- 3.1. Comandos e implementanción en XPRESS-MOSEL 21
- 3.2. Resultados computacionales 23

4. Conclusiones 29

A. Apéndice I 31
A.1. Modelo 1-PDTSP de ramificación y corte 31
A.2. Procedimiento 43

Bibliografía 47

Poster 49

Introducción

La presente memoria se titula **Problema del Viajante de Comercio con recogida y entrega de mercancía *(Pick-up and Delivery Travelling Salesman Problem (1-PDTSP))***. En ella procederemos a dar una concisa explicación sobre en qué consiste este proyecto de fin de grado, mostrando para ello una pequeña descripción de como se distribuyen, y los objetivos por los que se ha llevado a cabo.

El documento está estructurado en cuatro capítulos. En el Capítulo 1 expondremos una serie de conceptos teóricos necesarios para llevar acabo un buen desarrollo del proyecto, de manera que así se tendrá finalmente una buena comprensión de este trabajo. En el Capítulo 2 nos centramos, de una forma más concreta y extensa, en la definición del problema en el cual se ha basado este proyecto, el 1-PDTSP, así como, su formulación matemática general y las cinco variantes que citaremos. Posteriormente, en el Capítulo 3, se mostrarán los resultados computacionales obtenidos. Por último, en el Capítulo 4 se darán algunas conclusiones que se han logrado sacar en base a los resultados y las cuales contribuirán en la elaboración de próximos proyectos.

Motivación y Objetivos

Debido al gran interés mostrado por diversos sectores del mundo empresarial, como son los que mantienen una estrecha y directa relación con los procesos logísticos, de problemas de localización o de planificación de tareas, entre otros, vemos que aparece la necesidad de utilizar los modelos de la programación matemática y concretamente los modelos derivados de la optimización combinatoria, cuyo principal objetivo es optimizar un problema concreto o generalmente varios problemas en conjunto, generando y creando para ello algoritmos de resolución que aporten soluciones en tiempos computacionales razonables. Puesto

que sin dichos algoritmos tales problemas tendrían como inconveniente el hecho de que tras su resolución se pueda obtener un gran número de soluciones posibles.

Este hecho junto con la aspiración de ampliar los conocimientos ya adquiridos en el desarrollo de la asignaturas llamada *"Programación Combinatoria"* impartida en el Grado de Matemáticas de la Universidad de La Laguna han influido en la motivación a la hora de tomar la elección de afrontar y realizar el presente proyecto.

Una vez hubimos elegido el presente Trabajo Fin de Grado se plantearon la siguiente serie de objetivos que deberemos ver cumplidos durante el desarrollo y al final de este proyecto.

- Estudiar de manera más especifica algunos de los métodos y modelos matemáticos que existen en la actualidad y los cuales son empleados para resolver nuestro problema, el 1-PDTSP.
- Familiarizarnos con los programas informáticos que nos permitan programar los modelos para resolver dichos problemas de la manera más efectiva posible.
- Seguidamente y mediante los experimentos que se llevarán a cabo se pretende lograr resultados que corroboren el hecho de que tales métodos y modelos son de gran ayuda para conseguir resolver otros problemas que puedan afrontarse en posibles estudios futuros.

1

Fundamentos teóricos y notación

En este primer capítulo procedemos a hablar de los conceptos teóricos empleados en la resolución de nuestro problema. Comenzamos dando una pequeña definición sobre el concepto de Programación Matemática y de Optimización Combinatoria, así como los tipos de Programación Matemática que existen según sus principales características. Luego se explican brevemente los problemas de rutas, concretamente el Problema del Viajante de Comercio *(Traveling Saleman Problem o conocido también como TSP)* y los Problemas de rutas de vehículos *(Capacitated Vehicle Routing Problems o llamado también VRPs)*, y se exponen las variantes de estos dos problemas. Por último, explicaremos el método de resolución empleado para resolver y modelar nuestro problema en cuestión, dicho método se denomina Algoritmo de Ramificación y Corte. Y daremos una breve pero clara definición de las herramientas de las que hemos dispuesto.

1.1. Programación Matemática

La Programación Matemática forma parte de las Matemáticas Aplicadas y concretamente se encuentra dentro del área de conocimiento conocida como Investigación Operativa, y tiene como objetivo principal minimizar o maximizar, dependiendo del propósito que tengamos, una función objetivo satisfaciendo para ello una serie de restricciones, las cuales vienen formuladas por ecuaciones o inecuaciones. Matemáticamente y de forma general, este problema se puede representar de la siguiente manera:

$$\text{mín}\{f(x) : x \in S\} \tag{1.1}$$

donde $S \subseteq \mathbb{R}^n$ y $f : S \longrightarrow \mathbb{R}$, definiéndose S como el conjunto en el cual la función f alcanza el valor mínimo que se quiere obtener. A esta función f se le denomina función objetivo, al conjunto S se denomina región factible y a cada

uno de los elementos de este conjunto se le denomina solución factible. De este modo, también se pueden afrontar problemas de maximización puesto que si queremos maximizar dicha función en lugar de minimizarla solo tendríamos que tener el cuenta la siguiente equivalencia o igualdad:

$$\text{máx}\{f(x) : x \in S\} = -\,\text{mín}\{-f(x) : x \in S\} \tag{1.2}$$

A los problemas del tipo (1.1) se les denomina *problemas de optimización*, y tienen como objetivo encontrar una solución factible x^* de forma que $f(x^*)$ sea lo más pequeño posible. Esta búsqueda puede dar como resultados, las siguientes tres opciones:

- **Problema no factible:** este primer resultado posible se obtiene cuando no se encuentra ninguna solución factible para el problema, es decir, cuando $S = \emptyset$, en donde resultaría que $\text{mín}\{f(x) : x \in S\} = +\infty$.
- **Problema no acotado:** este resultado se obtiene cuando son encontradas soluciones que hacen descender a la función objetivo de forma infinita, es decir, cuando $\forall N \in \mathbb{R}$ se tiene que $\exists x \in S$ tal que $f(x) < N$, en donde se tendría que $\text{mín}\{f(x) : x \in S\} = -\infty$.
- **Problema con óptimo:** se obtiene este resultado cuando $\forall x \in S$ se tiene que $\exists x^* \in S$ tal que $f(x^*) \leq f(x)$. En dicho caso se tendrá que $\text{mín}\{f(x) : x \in S\} = f(x^*)$, y de esta manera x^* es denominada *solución óptima*.

Por otro lado, dependiendo de si nos centramos en las ecuaciones/inecuaciones o en las variables del problema se puede distinguir los siguientes tipos de Programación Matemática:

- En primer lugar, desde el punto de vista de la linealidad se tienen los siguientes tipos: en el caso de que las variables de la función objetivo y las ecuaciones o inecuaciones sean lineales se tiene la denominada **Programación lineal**, y en caso contrario, se tiene la **Programación no lineal**.
- En segundo lugar, según las características de las variables se tienen los siguientes tipos de programación matemática:
  - **Programación Lineal Continua:** Este tipo de programación es aquella en la cual la función objetivo f es una función lineal en las componentes $x \in S$ y donde estas componentes son las soluciones de un sistema lineal de inecuaciones. Además, una de las características principales de este tipo de programación es que todas las variables pueden tomar valores reales.
  - **Programación Lineal Entera:** Este segundo tipo se caracteriza porque por lo contrario los variables y las componentes $x \in S$ solo pueden tomar valores enteros.
  - **Programación Lineal Mixta:** Este tipo es una combinación de las dos anteriores puesto que se puede dar el caso en que se empleen variables continuas como enteras.

La Optimización Combinatoria es una parte importante dentro de la Programación Matemática puesto que se encarga del desarrollo de algoritmos con el fin de estudiar los problemas del tipo mín$\{f(x) : x \in S\}$, caracterizado por el hecho de que el conjunto de soluciones factibles es finito, aunque generalmente muy grande, es decir, $\mid S \mid < \infty$. Debido a que la mayoría de de los problemas de optimización combinatoria pueden resolverse como problemas de programación lineal entera, la optimización combinatoria y la programación lineal entera se encuentran bastante enlazadas entre sí.

1.2. Problemas de rutas

Dentro de la optimización combinatoria se encuentra un ámbito de investigación denominado Problemas de Rutas, del ingles *Routing Problems*. Donde se tiene como objetivo principal encargarse de la optimización de un conjunto de rutas de transporte que tienen que ser recorridas por uno o más vehículos, los cuales parten de un mismo depósito. Aquí explicaremos brevemente los dos problemas de rutas en los cuales hemos basado este documento y algunas de las variantes de dichos problemas.

1.2.1. Problema del Viajante de Comercio

Este problema conocido también como *Traveling Salesman Problem (TSP)*, es probablemente el problema más importante dentro de la Optimización Combinatoria puesto que ha sido muy estudiado debido a la dificultad a la hora de resolverlo a pesar de su comprensible definición, y puesto que es un problema muy general y de muchas aplicaciones prácticas.

El problema consiste en lo siguiente:

"Un viajante quiere visitar n ciudades una y sólo una vez cada una, empezando por una cualquiera de ellas y regresando al mismo lugar del que partió. Supongamos que conoce la distancia entre cualquier par de ciudades. ¿ De qué forma debe hacer el recorrido si pretende minimizar la distancia total?. " (Stockdale, [5])

Históricamente, y tal como lo conocemos, este problema tuvo su primera aparición en 1930, a partir de la cual muchos investigadores intentaron emplear y aportar distintas herramientas con las que resolver más eficientemente este problema. En la Tabla 1.1 se muestran algunos de los más destacados:

Número de ciudades	Autores	Año
42	Dantzig, Fulkerson, Johnson	1945
48	Held,Karp	1962
57	Karg,Thompson	1964
64	Held,Karp [8]	1971
80	Helbig, Hanon, Krarup	1974
100	Camerini, Fratta, Maffioli	1975
120	Grötschel [10]	1980
318	Crowder, Padberg, Hong [11]	1980
532	Padberg, Rinaldi [12]	1987
666	Grötschel, Holland [13]	1991
2392	Padberg, Rinaldi [14]	1991
7397	Applegate, Bixby, Chvátal, Cook [15]	1994
13509	Applegate, Bixby, Chvátal, Cook	1998
15112	Applegate, Bixby, Chvátal, Cook	2001
24978	Applegate, Bixby, Chvátal, Cook and Helsgaun	2004
33810	Cook, Espinoza and Goycoolea [16]	2005
85900	Applegate, Bixby, Chvátal and Cook [17]	2006

Tabla 1.1. Mayor tamaño de problemas resueltos

Dicho con otras palabras, se tiene un viajante que debe visitar una cierta cantidad n de ciudades, pasando únicamente una vez por cada una de ellas, y el cual está sujeto a una serie de restricciones:

- El circuito que se genere será un circuito cerrado puesto que se debe prefijar una de las ciudades como el punto de partida y el punto de llegada en dicho recorrido.
- Inicialmente, tanto las distancias entre las ciudades como los costos de ir de una ciudad a otra son datos conocidos.
- El grafo que se obtendrá y que ha sido generado por las distintas ciudades podrá ser de dos tipos:
  - Cuando las conexiones entre las ciudades no tienen un dirección prefijada entonces se denominará grafo no dirigido.
  - Por lo contrario, si se conocen la dirección entre todas las conexiones entre las ciudades entonces se tendrá que el grafo es un grafo dirigido.

Finalmente, y siempre que se hayan cumplido las condiciones anteriores, el objetivo será encontrar un ruta con la cual el coste final sea mínimo.

El resultado que se obtendría una vez resuelto el problema es un circuito hamiltoniano o también llamado ciclo simple.

Definición (Circuito Hamiltoniano): *Un circuito o camino es hamiltoniano si es un circuito que visita cada vértice una única vez, exceptuando el vértice de partido del cual sale y al cual tiene que regresar al final del circui-*

to. Si un grafo contiene un circuito hamiltoniano entonces es denominado grafo hamiltoniano.

Matemáticamente hablando el problema viene definido por la siguiente notación:

Sea $G = (V, A)$ un grafo dirigido, donde $V = \{1, \ldots, n\}$ es el conjunto de nodos y $A \subseteq \{(i,j) : \ i,j \in V, \ i \neq j\}$ el conjunto de arcos. Para cada arco $a \in A$ denotamos por c_a al costo de pasar por dicho arco. Por otro lado, para cada $S \subseteq V$ representamos por $\delta^+(S) = \{(i,j) \in A : i \in S, j \notin S\}$ y por $\delta^-(S) = \{(i,j) \in A : j \in S, i \notin S\}$ al conjunto de arcos que salen y entran en el conjunto S, y particularmente representamos a los arcos que salen y entran al nodo $i \in V$ como $\delta^+(\{i\})$ y $\delta^-(\{i\})$, respectivamente. Por último, denotamos por $A(S) = \{(i,j) \in A : i,j \in S\}$ al conjunto de arcos pertenecientes a la región factible.

Para poder modelar y posteriormente resolver dicho problema definimos la siguiente variable de decisión:

$$x_a = \begin{cases} 1 \text{ si } a \in T, \text{ con } T \equiv \text{circuito del grafo} \\ 0 \text{ en otro caso} \end{cases}$$

Con la notación y la variable de decisión definida anteriormente el modelo de Programación Lineal Entera para el TSP se expresa de la siguiente manera:

$$\text{mín} \sum_{a \in T} c_a x_a \tag{1.3}$$

sujeta a:

$$\sum_{a \in \delta^+(\{i\})} x_a = 1 \qquad \text{para todo } i \in V \tag{1.4}$$

$$\sum_{a \in \delta^-(\{i\})} x_a = 1 \qquad \text{para todo } i \in V \tag{1.5}$$

$$\sum_{a \in A(S)} x_a \leq \mid S \mid -1 \qquad \text{para todo } S \subset V \ , \ S \neq \emptyset \tag{1.6}$$

$$x_a = \{0,1\} \qquad \text{para todo } a \in A \tag{1.7}$$

La ecuación (1.3) representa la función objetivo. Las restricciones (1.4) y (1.5) imponen que de cada nodo debe tanto salir como entrar exactamente un arco. Las restricciones (1.6) impide que se formen subciclos. Y la restricción (1.7) indica que la variable es binaria.

Variantes del TSP

- **Problema del viajante de comercio con ventanas de tiempo:** conocido en inglés como *Traveling Salesman Problem with Time Windows*, es el

problema en el cual cada ciudad y cada cliente tienen un tiempo mínimo y un tiempo máximo para ser visitado.

- **Problema del viajante de comercio generalizado:** *(Generalized Traveling Salesman Problem)*, las ciudades están agrupadas en regiones y se tiene que visitar como mínimo una ciudad de cada región de tal forma que se minimice el coste.
- **Problema del Ciclo Simple:** *(Simple Cycle Problem)*, además de tener unos costos de ir de una ciudad a otra se tiene también unos beneficios por visitar una ciudad, lo que tiene como consecuencia que no sea necesario que en la región factible estén incluidas todas las ciudades.
- **Problema del viajante de comercio con múltiples viajantes:** en esta variante existe una cantidad determinada de viajantes y tiene como condición o restricción principal que ningún viajante puede visitar una ciudad que ya ha sido visitada por otro viajante.
- **Problema del viajante de comercio con recogida y entrega de mercancías:** *(Pick-up and Delivery Traveling Salesman Problem (PDTSP))*, cada ciudad recibe o proporciona una cierta cantidad de producto transportada mediante el uso de un único vehículo con capacidad limitada. Esta variante del Problema del Viajante de Comercio será la que posteriormente, concretamente en el Capítulo 2 de este documento, explicaremos con más detenimiento.

1.2.2. Problemas de rutas de vehículos

Este otro tipo de problemas de rutas también conocido como *Vehicle Routing Problems (VRPs)* es una generalización del problema del viajante de comercio. Se conocen las demandas de los clientes y se tiene como restricción principal que los vehículos solo puedan visitar a los clientes una única vez. Y tiene como principal objetivo encontrar un conjunto de rutas las cuales salgan inicialmente y terminen siempre en el depósito (o los depósitos), de tal forma que se minimice el coste total.

Variantes del VRP

- **Problema de rutas de vehículos con capacidad limitada *(CVRP)*:** este problema viene determinado por un conjunto de rutas que empiezan y terminan en un mismo depósito, en donde cada ruta es realizada por un único vehículo y mediante un solo recorrido, y a través de los cuales finalmente se debe minimizar los costes globales de transporte, satisfaciendo una serie de restricciones:
  - Si del deposito salen una cantidad m de vehículos, idénticamente iguales, entonces después tiene que entrar exactamente m vehículos.

- A cada cliente se debe entrar, y de cada cliente se debe salir una única vez.
- Cada cliente debe recibir exactamente la demando que solicitó.
- A través de cada conexión se transportará como máximo la carga máxima que se permite ser transportada el vehículo. Dado esto se asume que la suma de todas las demandas de los clientes debe ser menor o igual que el producto del número de vehículos por su carga máxima permitida, es decir, $\sum_{i=2}^{n} d_i \leq mQ$, donde d_i es la demanda solicitada por el cliente i, m es el número de vehículos que hay y Q es la carga máxima que puede transportar cada vehículo.

Dada esta definición del *CVRP*, introducimos la siguiente notación. Sea un grafo $G = (V, A)$ el cual consideraremos dirigido, donde $V = \{1, \ldots, n\}$ es el conjunto de localizaciones, en el cual 1 es el deposito y $\{2, \ldots, n\}$ el conjunto de clientes, y $A = \{a = (i, j) : i, j \in V \text{ y } i \neq j\}$ es el conjunto de arcos. Se conocen los costos c_{ij} de ir desde $i \in V$ hasta $j \in V$. Y por último, sean d_i, m, Q como hemos definido anteriormente. Para modelar el problema introducimos las siguientes variables de decisión:

$$x_a = \begin{cases} 1 \text{ si el arco } a \in A \text{ está en la ruta de algún vehículo} \\ 0 \text{ en otro caso} \end{cases}$$

$$f_a \equiv \text{ carga transportada por el vehículo al pasar por el arco } a \in A$$

De esta manera el modelo del Problema de Rutas de Vehículos con Capacidad limitada *(CVRP)* será el siguiente:

$$\text{mín} \sum_{a \in A} c_a x_a \tag{1.8}$$

sujeto a:

$$\sum_{a \in \delta^+(\{i\})} x_a = \sum_{a \in \delta^-(\{i\})} x_a = 1 \qquad \text{para todo } i \in V \diagdown \{1\} \tag{1.9}$$

$$\sum_{a \in \delta^+(\{1\})} x_a = \sum_{a \in \delta^-(\{1\})} x_a \leq m \tag{1.10}$$

$$\sum_{a \in \delta^+(\{i\})} f_a - \sum_{a \in \delta^-(\{i\})} f_a = d_i \qquad \text{para todo } i \in V \diagdown \{1\} \tag{1.11}$$

$$0 \leq f_a \leq Q x_a \qquad \text{para todo } a \in A \tag{1.12}$$

$$x_a = \{0, 1\} \qquad \text{para todo } a \in A \tag{1.13}$$

La formula (1.8) es la función objetivo. Las restricciones (1.9)–(1.12) imponen que se cumplan todas las condiciones expuestas anteriormente en la explicación del problema. Y la restricción (1.13) indica que la variable sea binaria.

Otra forma posible de impedir que se produzcan subciclos es sustituir las restricciones (1.11) y (1.12) por las siguientes restricciones (1.14):

$$\sum_{a \in A(S)} x_a \leq \mid S \mid - \left\lceil \sum_{i \in V} d_i / Q \right\rceil \text{ para todo } S \subset V \tag{1.14}$$

- **Problema de rutas de vehículos con ventanas de tiempo:** Esta variante del *VRP* viene dada por las mismas variables de entrada y de salida que la variante anterior y por una nueva serie de variables de entrada que vienen definidas de la siguiente manera: sean $[a_i, b_i]$ la ventana de tiempo dentro de la cual se debe comenzar el transporte, t_{ij} el tiempo que lleva ir desde i hasta j, p_i el tiempo que tarda el cliente i en procesar la mercancía. Definimos una nueva variable z_i para cada cliente $i \in V$ que se corresponde con el tiempo de llegada de la mercancía al cliente i, con $i, j \in V$, y sea M una constante suficientemente grande que está definida como el tiempo máximo. Y las cuales conllevan a que se cumplan las nuevas restricciones:

$$a_i \leq z_i \leq b_i \text{ para todo } i \in V \tag{1.15}$$

$$z_j \geq z_i + p_i + t_{ij} - M(1 - x_{ij}) \text{ para todo } i, j \in V \text{ , } i \neq j \text{ y } j \neq 1 \tag{1.16}$$

La restricciones (1.15) imponen que el tiempo de llegada de la mercancía esté dentro de la ventana de tiempo establecida. Y la restricciones (1.16) acotan inferiormente el tiempo de llegada de la mercancía al cliente.

- **Problema de rutas de vehículos con múltiples depósitos (*Multi-depot Vehicle Routing Problem (MDVRP)*):** Esta tercera variante del *VRP* viene dada por las mismas variables de entrada y de salida que la primera variante del *VRP* expuesta, el *CVRP*, exceptuando tres cambios que se producen: el primero es que la variable de decisión f_a que desaparece y con ella las restricciones que debía cumplir, en segundo lugar el conjunto V que cambia por el hecho de que ahora habrán un número p de depósitos y se define de la siguiente manera $V = \{1, \ldots, p, p+1, \ldots, p+n\}$ donde $D = \{1, \ldots, p\}$ es el conjunto de depósitos y $\{p+1, \ldots, p+n\}$ que es el conjunto de clientes, y en tercer lugar y debido a la cantidad p de depósitos hay ahora, ahora habrá una cantidad $m_i = m_1, \ldots, m_p$ de vehículos en cada depósito $i \in D$. Respecto a las restricciones, además de desaparecer como ya comentamos la restricciones relacionadas con la variable f_a, se produce un cambio en la restricción (1.9) y dicho cambio es el siguiente:

$$\sum_{a \in \delta^+(\{i\})} x_a = \sum_{a \in \delta^-(\{i\})} x_a \leq m_i \text{ para todo } i \in D \tag{1.17}$$

1.3. Método de resolución: Algoritmo de ramificación y corte

Esta técnica es una combinación de otras dos técnicas, *el algoritmo de hiperplanos de corte*, la cual se encargará de resolver los problemas que se vayan generando, y *el algoritmo de ramificación y acotación (branch-and-bound)* cuya función es resolver problemas de programación matemática lineal cuyas variables son enteras.

A continuación se muestra de forma resumida el procedimiento que lleva a cabo este algoritmo o método de ramificación y corte:

Definimos para ello la siguientes notaciones:

- El conjunto x_e con $e \in I$, donde $I \equiv$ conjunto de índices, como el conjunto de variables enteras.
- $EP \equiv$ Problema entero
- $z_{EP} \equiv$ Valor óptimo del problema entero
- $RP \equiv$ Relajación lineal del problema entero
- $z_{RP} \equiv$ Valor óptimo del problema relajado
- $m \equiv$ número de restricciones existentes en EP

El primer paso que se lleva a cabo es eliminar las restricciones de integrabilidad, de manera que así obtengamos la relajación lineal del problema de programación lineal entera.

De este modo, en el caso de tener que minimizar la función objetivo se tendrá que $z_{RP} \leq z_{EP}$, es decir, se tendrá que el valor óptimo del problema relajado es una cota inferior del valor óptimo del problema entero.

En el siguiente paso del algoritmo se procede según cual sea el número de restricciones, m, que existen en el problema entero EP.

Dado que este número de restricciones puede ser muy grande, denotaremos por RP_∞ al problema RP que contenga una cantidad grande de restricciones y por RP_h, con $h \geq 0$ al problema RP que contenga una cantidad razonable de restricciones, y el cual resolveremos a continuación, obteniéndose una solución óptima a la que vamos a denotar como $x^*_{RP_h}$. Hecho esto, se comprueba si $x^*_{RP_h}$ es o no factible para el problema EP. En el caso de obtener que es factible se tendrá que dicha solución $x^*_{RP_h}$ es la solución óptima de EP y de este modo estaría resuelto el problema entero puro. Si por lo contrario se obtiene que no es factible entonces dicho algoritmo de separación nos devolverá, en caso de haberla, la restricción que ha sido violada y la cual se añadirá al problema RP_h de forma que se obtendrá un nuevo problema relajado RP_{h+1}. Pero en el caso de no haber ninguna restricción violada el algoritmo nos comunicará que todas las restricciones han sido satisfechas.

Mediante este procedimiento se ha de obtener una solución óptima para el problema RP_∞ a la que denotaremos por $x^*_{RP_\infty}$.

Finalmente, si dicha solución óptima es entera entonces será la solución óptima para el problema entero puro, EP. Sin embargo, si dicha solución óptima no es entera se procederá a emplear el algoritmo de ramificación de la siguiente manera. Definimos x_e como una variable fraccionaria de $x^*_{RP_\infty}$ y construimos los siguientes dos problemas, $P1$ como el problema relajado con la restricción $x_e \geq \lceil x^*_e \rceil$ y $P2$ como el problema relajado con la restricción $x_e \leq \lfloor x^*_e \rfloor$. Una vez hecho esto tendremos formado un árbol en el que cada nodo se corresponde con un problema relajado. Entonces se comprueba si se cumple o no las siguientes condiciones: la solución obtenida x^*_e sea entera y dicha solución sea estrictamente mayor que z_{EP}. Si se tiene que se han cumplido entonces se obtiene que x^*_e es la solución de EP, pero si de lo contrario no se cumplen entonces se vuelve a ramificar respecto a cada nodo que se ha obtenido a partir de cada uno de los problemas $P1$ y $P2$.

1.4. Material y herramientas de trabajo

A continuación se presentan las distintas herramientas que se tuvieron en cuenta para implementar este proyecto. Seguidamente, diremos cual fue la elección final y daremos una descripción más detallada de dicho programa.

Las tres opciones que se tuvieron en cuenta en un comienzo fueron las siguientes:

- CPLEX: es un optimizador/compilador que emplea como lenguaje de programación el lenguaje OPL. Este tipo de lenguaje se emplea principalmente para representar problemas de optimización y el cual aporta facilidades en la organización de los datos, posee una buena conección con Bases de Datos y Excel, además tiene la capacidad de resolver algoritmos mediante distintos tipos de programación, entre los que se encuentran la programación lineal y entera, la programación basada en restricciones o los modelos cuadráticos.
- GUSEK *(GLPK Under Scite Extended Kit)*: se trata de entorno gráfico gratuito, que usa como herramienta principal el GLPK *(GNU Linear Programming Kit)* para resolver modelos matemáticos lineales. El lenguaje de modelado que posee se denomina GMPL (GPL) debido a que es un lenguaje de programación muy flexible que se encuentra basado en C++.
- Xpress: este es un compilador empleado de manera principal para resolver Problemas de Programación Lineal *(Linear Programming, LP)* y de Programación Lineal Entera Mixta *(Mixed Integer Linear Programming)*. Xpress tiene como lenguaje de modelado el lenguaje denominado Xpress-Mosel.

Al inicio de este proyecto y en un primer instante, consideramos hacer uso del compilador CPLEX puesto que se ajustaba mejor a nuestras necesidades, sin embargo tuvimos que descartar esta opción debido a que nos encontramos

con las limitaciones que presentaba la versión estudiantil. Finalmente, entre las otras dos alternativas, optamos por el Xpress-Mosel, el cual describiremos más específicamente a continuación.

Mosel es un entorno que proporciona un lenguaje de programación y a la vez un modelo con el objetivo de modelar y resolver problemas. Ofrece las condiciones necesarias para declarar y manipular problemas, variables de decisión, restricciones, y diversos tipos de datos y de estructuras, como son por ejemplo los conjuntos y las matrices. También proporciona un conjunto de funcionalidades muy completo en el que admite todas las construcciones de selección y bucles. Además, dentro del lenguaje de programación Mosel no existe separación entre las declaraciones del modelado y un procedimiento que resuelva realmente el problema. Otras dos de sus características importantes son las siguientes: la sintaxis empleada ha sido diseñada de modo que el conjunto de construcciones y de palabras reservadas se mantenga deliberadamente pequeño con el fin de que así este lenguaje sea fácil de aprender, y ofrece una interfaz dinámica estructurada por módulos de manera que cada uno de estos módulos incluye su propio conjunto de procedimientos y funciones, lo cual proporciona a Mosel una ampliación de su vocabulario y capacidades. Estas características hacen que en Mosel sea posible programar complejos algoritmos para la resolución de problemas.

2

Problema del Viajante de Comercio con recogida y entrega de mercancía

En este segundo capítulo nos centraremos concretamente en el *Problema del Viajante de Comercio con recogida y entrega de mercancía*. Se define el problema, se ven aplicaciones prácticas y finalmente se muestran cinco formas distintas de resolverlo mediante modelos matemáticos diferentes.

2.1. Introducción

El Problema del Viajante de Comercio con recogida y entrega de mercancía llamado también *Pick-up and Delivery Traveling Salesman Problem (1-PDTSP)* viene definido por las siguientes características:

- Se tienen un conjunto de n localizaciones (o estaciones) de las cuales se recoge o a las cuales se entrega la mercancía.
- Hay un vehículo que posee una capacidad máxima de carga, el cual tiene que mover la mercancía de una localización a otra de forma que en cada una de las localizaciones se quede exactamente la cantidad de mercancía que solicitaron. Dicha cantidad es mayor que 0 en el caso de que el vehículo recoja mercancía de la localización y es negativa en el caso de que la entregue. Se supone el hecho de que la mercancía que es entregada por una localización (se recoge mercancía) puede ser entregada en otra localización (se entrega mercancía), dado esto asumimos que la contabilización de todos estas cantidades de mercancía, entregadas y recogidas, sea nula, es decir, $\sum_{i=1}^{n} d_i = 0$, donde d_i es el número de unidades de producto que recoge o entrega el cliente i, respectivamente si $d_i > 0$ o $d_i < 0$.
- Además se conocen el costo de transportar mercancía de una localización a otra.
- Por último se tiene como objetivo minimizar el coste global de forma que se visite cada localización una única vez.

2.2. Modelos matemáticos para el 1-PDTSP

Ahora exponemos la notación que emplearemos posteriormente para poder dar una primera formulación del problema. Decimos esto puesto que en el apartado siguiente se mostrarán cuatro variantes del modelo, las cuales varían en cuanto a las variables que se definen o el algoritmo del que se hace uso para su resolución. Entonces, la notación general que tendremos para el problema es la siguiente:

- $G(V, A) \equiv$ grafo, el cual supondremos que es dirigido, donde:
  - $V = \{1, \ldots, n\}$ es conjunto de localizaciones, 1 es el depósito y $\{2, \ldots, n\}$ es el conjunto de clientes
  - $A = \{(i, j) : i, j \in V \text{ con } i \neq j\} \equiv$ conjunto de arcos
- c_{ij} es el costo de ir desde la localización i hasta la localización j, con $i, j \in V$
- Q es la capacidad máxima de carga del vehículo
- d_i es el número de unidades de producto que recoge el cliente. i, Si $d_i < 0$ significará que el cliente i entrega $-d_i$ unidades de producto

A continuación mostraremos y explicaremos detalladamente cinco variantes de este modelo, para uno de los cuales se aplicará a medida *el algoritmo de ramificación y cortes* que ya hemos citado y explicado en el Capítulo 1.

2.2.1. Modelo con restricciones de subciclos

En primer lugar, definimos las variables de decisión necesarias para luego efectuar la correcta formulación y resolución de esta primera variante del problema:

$$x_a = \begin{cases} 1 \text{ si el vehículo se desplaza desde la localización } i \text{ hasta } j, \text{ con } i, j \in V \\ 0 \text{ en otro caso} \end{cases}$$

$$f_{ij} \equiv \text{ cantidad de mercancía (carga) que transporta el vechículo}$$

Por tanto, el modelo matemático del *Problema del Viajante de Comercio con recogida y entrega de mercancía (1-PSTSP)* quedará formulado del modo siguiente:

$$\text{mín} \sum_{a \in T} c_a x_a \tag{2.1}$$

sujeta a:

$$\sum_{a\in\delta^+(\{i\})} x_a = \sum_{a\in\delta^-(\{i\})} x_a = 1 \qquad \text{para todo } i \in V \tag{2.2}$$

$$\sum_{a\in\delta^+(\{i\})} f_a - \sum_{a\in\delta^-(\{i\})} f_a = d_i \qquad \text{para todo } i \in V \tag{2.3}$$

$$0 \leq f_a \leq Q x_a \qquad \text{para todo } a \in A \tag{2.4}$$

$$\sum_{a\in A(S)} x_a \leq |S| - 1 \qquad \text{para todo } S \subset V \text{ donde } S \neq \emptyset \tag{2.5}$$

$$x_a = \{0, 1\} \qquad \text{para todo } a \in A \tag{2.6}$$

La fórmula (2.1) es la función objetivo del problema. Las restricciones (2.2) imponen que a cada localización se la visite exactamente una única vez. Las restricciones (2.3) mantienen la conservación de la carga en el sistema. Las restricciones (2.4) imponen que por cada arco la carga transportada no supere la cantidad máxima de carga permitida del vehículo. Las restricciones (2.5) impiden que que se formen subciclos imponiendo que el número total de arcos que hay en cada posible ruta, $S \subset V$, sea siempre menor o igual que el número de localizaciones que constituye dicha ruta menos 1. Y por último, las restricciones (2.6) imponen que la variable x_a sea binaria.

2.2.2. Modelo 1-PDTSP carga-flujo

Esta segunda variante surge de añadir a la formulación general del modelo, expuesta en la sección anterior, la variable de decisión g_{ij} definida como:

$$g_{ij} \equiv \text{ Carga de un flujo ficticio}$$

Donde dicho flujo ficticio ha sido creado con el propósito de evitar que se produzcan subciclos, imponiéndole para ello que en cada cliente $i \in V \setminus \{1\}\}$ se quede exactamente una unidad de dicha carga.

De modo que esta variante a la que hemos denominado 1-PDTSP carga-flujo queda formulada de la siguiente manera:

$$\text{mín} \sum_{a\in T} c_a x_a \tag{2.7}$$

sujeta a:

$$\sum_{a\in\delta^+(\{i\})} x_a = \sum_{a\in\delta^-(\{i\})} x_a = 1 \qquad \text{para todo } i \in V \tag{2.8}$$

$$\sum_{a\in\delta^+(\{i\})} f_a - \sum_{a\in\delta^-(\{i\})} f_a = d_i \qquad \text{para todo } i \in \tag{2.9}$$

$$0 \leq f_a \leq Qx_a \qquad \text{para todo } a \in A \tag{2.10}$$

$$\sum_{a\in\delta^+(\{i\})} g_a - \sum_{a\in\delta^-(\{i\})} g_a = 1 \qquad \text{para todo } i \in V \setminus \{1\} \tag{2.11}$$

$$0 \leq g_a \leq (n-1)x_a \qquad \text{para todo } a \in A \tag{2.12}$$

$$x_a = \{0,1\} \qquad \text{para todo } a \in A \tag{2.13}$$

Donde la restricción (2.11) que impone el sentido que tienen los arcos $a \in A$, es decir, que se transporte la mercancía desde el cliente i hacia el cliente j ó viceversa, pero no en ambos sentidos. Y la restricción (2.12) impone que todos los clientes tienen que ser visitados.

2.2.3. Modelo 1-PDTSP multiflujo

Esta tercera variante surge de añadir a la formulación general del modelo, expuesta en la sección anterior, la variable de decisión g_{ij}^k definida como:

$$g_{ij}^k = \begin{cases} 1 \text{ si se quiere llegar al destino } k \text{ se lleva por el arco } (i,j) \\ \text{, con } i,j \in V,\ k \in V \setminus \{1\} \\ 0 \text{ en otro caso} \end{cases}$$

De forma que esta variante a la que hemos denominado 1-PDTSP multiflujo queda formulado de la siguiente manera:

$$\text{mín} \sum_{a\in T} c_a x_a \tag{2.14}$$

sujeta a:

$$\sum_{a\in\delta^+(\{i\})} x_a = \sum_{a\in\delta^-(\{i\})} x_a = 1 \qquad \text{para todo } i \in V \tag{2.15}$$

$$\sum_{a\in\delta^+(\{i\})} f_a - \sum_{a\in\delta^-(\{i\})} f_a = d_i \qquad \text{para todo } i \in \tag{2.16}$$

$$0 \leq f_a \leq Qx_a \qquad \text{para todo } a \in A \tag{2.17}$$

$$\sum_{j\in V, j\neq 0} g_{1j}^k = 1 \qquad \text{para todo } k \in V \setminus \{1\} \tag{2.18}$$

$$g^k_{j1} = 0 \qquad \text{para todo } j,k \in V \setminus \{1\} \tag{2.19}$$

$$\sum_{j \in V, j \neq k} g^k_{jk} = 1 \qquad \text{para todo } k \in V \setminus \{1\} \tag{2.20}$$

$$g^k_{kj} = 0 \qquad \text{para todo } j,k \in V \setminus \{1\}, j \neq k \tag{2.21}$$

$$\sum_{j \in V, j \neq i} g^k_{ji} - \sum_{j \in V, j \neq i} g^k_{ij} = 0 \qquad \text{para todo } i \in V \setminus \{1\}, i \neq k \tag{2.22}$$

$$g^k_{ij} \leq x_{ij} \qquad \text{para todo } i,j,k \in V \setminus \{1\} \tag{2.23}$$

$$x_a = \{0,1\} \qquad \text{para todo } a \in A \tag{2.24}$$

Donde las restricciones (2.18) imponen que desde el deposito sale una unidad de producto ficticio $k \in V \setminus \{1\}$. Las restricciones (2.19) imponen que al deposito llegan 0 unidades de producto $k \in V \setminus \{1\}$. Las restricciones (2.20) y (2.21) imponen que al nodo k llega una unidad de producto k y salen 0 unidades de producto k, respectivamente. La restricción (2.22) fijan que cuando el vehículo atraviesa un cliente destino del k se mantiene la carga del producto ficticio k. Por último, la restricción (2.23) impone que la variable g^k_{ij} sea menor o igual que x_{ij}.

2.2.4. Modelo 1-PDTSP con variable potencial

Esta cuarta variante surge de añadir a la formulación general del modelo, expuesta en la sección anterior, la variable de decisión u_i definida como:

$$u_i = \text{posición del cliente } i \text{ en la solución, para todo } i \in V \setminus \{1\}$$

De modo que esta variante a la que hemos denominado 1-PDTSP con variable potencial queda formulada de la siguiente manera:

$$\text{mín} \sum_{a \in T} c_a x_a \tag{2.25}$$

sujeta a:

$$\sum_{a \in \delta^+(\{i\})} x_a = \sum_{a \in \delta^-(\{i\})} x_a = 1 \qquad \text{para todo } i \in V \tag{2.26}$$

$$\sum_{a \in \delta^+(\{i\})} f_a - \sum_{a \in \delta^-(\{i\})} f_a = d_i \qquad \text{para todo } i \in V \tag{2.27}$$

$$0 \leq f_a \leq Q x_a \qquad \text{para todo } a \in A \tag{2.28}$$

$$u_j \geq u_i + x_{ij} - (n-2)(1-x_{ij}) + (n-3)x_{ji} \qquad \text{para todo } i,j \in V \tag{2.29}$$

$$x_a = \{0,1\} \qquad \text{para todo } a \in A \tag{2.30}$$

Donde la restricción (2.29) impone el que si $x_{ij} = 1$ entonces $u_j \geq u_i + 1$ y si no es así que el miembro del lado derecho de la desigualdad sea lo suficiente pequeño para que siempre se verifique. El último sumando del lado derecho de la desigualdad nos permite fortalecer la desigualdad en caso de que $x_{ji} = 1$.

2.2.5. Modelo 1-PDTSP con restricciones de capacidad

Esta variante surge de eliminar las variables f_{ij}, que representan la cantidad de mercancía que transporta el vehículo entre las localizaciones i y j, y las restricciones (2.3) y (2.4), y de modificar la restricción (2.5) de modo que se tenga en cuenta también cuando la capacidad del vehículo es sobrepasada. De este modo el modelo que se obtiene es el siguiente:

$$\text{mín} \sum_{a \in T} c_a x_a \tag{2.31}$$

sujeta a:

$$\sum_{a \in \delta^+(\{i\})} x_a = \sum_{a \in \delta^-(\{i\})} x_a = 1 \qquad \text{para todo } i \in V \tag{2.32}$$

$$\sum_{a \in A(S)} x_a \leq |S| - r(S) \qquad \text{para todo } S \subseteq V,\ S \neq \emptyset \tag{2.33}$$

$$x_a = \{0, 1\} \qquad \text{para todo } a \in A \tag{2.34}$$

Donde la restricción (2.33), en la cual $r(S) = \text{máx}\{1, \left\lceil \frac{|\sum_{i \in S} d_i|}{Q} \right\rceil\}$, tiene como objetivo evitar que se formen subciclos y que se conserve la capacidad máxima del vehículo, imponiendo para ello que el número total de arcos que hay en cada posible ruta, $S \subset V$, sea siempre menor o igual que el número de localizaciones que constituye dicha ruta menos el máximo entre 1 y la el cociente $\left\lceil \frac{|\sum_{i \in S} d_i|}{Q} \right\rceil$ formado por el sumatorio de las demandas de los clientes y la cantidad máxima de carga que puede transportar el vehículo.

Dado que el número de restricciones (2.33) es exponencial, por lo que no se pueden dar directamente para resolver a un programa, veremos dos maneras de resolverlo.

La primera es aquella en la que se resuelve el modelo sin restricciones (2.33) y una vez resuelto se verifica si hay alguna restricción (2.33) que haya sido violada, en cuyo caso se insertan al problema y posteriormente se vuelve a resolver. A esta primera forma la hemos denominado 1-PDTSP capacity.

En cambio, la segunda versión verifica si hay alguna restricción (2.33) violada sin la necesidad de que de forma previa se tenga que resolver el problema y comprobar que su solución sea entera, sino que basta con tener una solución fraccionaria de dicho problema. Esta importante labor de hallar dicha solución

fraccionaria es la que se lleva a cabo mediante el *algoritmo de Ramificación y Corte*, que hemos explicado en el capítulo anterior, concretamente en el apartado 1.3.

3

Implementación en Xpress-Mosel y resultados computacionales

Uno de los aspectos más difíciles de la realización de este trabajo ha sido la implementación de los modelos, los cuales ya hemos explicado detalladamente en el capítulo anterior. Especial dificultad ha tenido el modelo *1-PDTSP de Ramificación y Corte* debido a ciertos aspectos de la programación, la cual ya hemos explicado en el primer capítulo y se ha elegido realizar mediante el programa Xpress y su lenguaje Mosel, que hubo que estudiar previamente, y que detallaremos posteriormente en este capítulo. Finalmente, se expondrán los resultados obtenidos, concretamente los tiempos computacionales que han empleado en su ejecución cada modelo para varios ejemplos de distintos tamaños n y capacidades Q.

3.1. Comandos e implementanción en XPRESS-MOSEL

Como ya hemos comentado en la introducción de este capítulo, ha sido necesario realizar un estudio de características avanzadas del lenguaje Mosel puesto que para la correcta ejecución de los programas correspondientes a los modelos 1-PDTSP con restricciones de subciclos y 1-PDTSP de Ramificación y Corte, y de forma más notable para el segundo de dichos modelos, se ha tenido que hacer uso de varios comandos esenciales. Dichos comandos serán presentados y explicados a continuación:

Setcallback

Las *setcallback* son el recurso utilizado por Mosel con el fin de realizar las llamadas de forma recursiva a lo largo del proceso de búsqueda de las funciones y procedimientos que han sido definidos por el usuario durante el método de Ramificación y Corte. Esta llamada posee el siguiente tipo de sintaxis:

Setcallback(“. . . *tipo de llamada recursiva . . .* ”, “. . . *nombre de la función o procedimiento llamado . . .* ”)

Existen varios tipos de llamadas iterativas, cada uno de los cuales tienen una labor diferente según a que determinada función o procedimiento estén asociados. Dichos tipos fueron estudiados a partir de los manuales de Mosel y de Xpress, [6]. A continuación, veamos los dos tipos que hemos empleado en nuestro caso:

- **setcallback(XPRS_CB_PREINTSOL, “cb_entera(isheur:boolean, cutoff:real)”)**
  Una vez ha sido encontrada una solución entera este procedimiento recursivo se encarga de efectuar la llamada de la función correspondiente para verificar que dicha solución entera es válida. Este tipo de procedimiento puede ser bastante útil a la hora de encontrar las soluciones enteras puesto que en ese momento el método de ramificación y corte no es llamado y en el caso de que la solución encontrada no sea factible se pueda excluir.
- **setcallback(XPRS_CB_CUTMGR, “cb_fraccional:boolean”)**
  Para que el método de Ramificación y Corte sea implementado de un modo correcto esta función tiene la tarea de controlar los cortes recursivos procediendo de la siguiente manera. Una vez se ha llegado a un nodo del árbol de ramificación, es decir, a un problema relajado, el resolutor llama a la función “ cb_fraccional ”, la cual se encarga de resolver el problema de separación. En el caso de que se haya encontrado y añadido un nuevo corte no se ramifica hacia otro nodo sino que la función devuelve al problema general un parámetro booleano, y posteriormente resuelve nuevamente el problema con el corte añadido y vuelve a llamar a la función “ cb_fraccional ”. En caso contrario, como se ha indicado anteriormente, si no ha sido añadido un corte entonces la función procede a ramificar hacia otro nodo. Todo este proceso se repite de forma iterativa hasta que no se encuentra ningún corte que añadir al problema general, y una vez llegado a dicho punto, el programa ramifica hacia otro nodo.

Addcuts

Los *addcuts* es una rutina encargada de añadir el corte o los cortes al nodo actual y a todos sus descendientes. La sintaxis que posee esta rutina es la siguiente:

Addcuts(“. . . *identificación del corte . . .* ”, “. . . *tipo de desigualdad . . .* ”, “. . . *corte . . .* ”)

Donde la identificación del corte se efectúa mediante el uso de un número entero, y los tipos de desigualdad pueden ser los siguientes: para la desigualdad mayor o igual (CT_GEQ), para la desigualdad menor o igual (CT_LEQ) y para la igualdad (CT_EQ).

3.2. Resultados computacionales

A continuación mostraremos los resultados computacionales obtenidos mediante distintas tablas, pero antes de ello expondremos las principales características del sistema en el cual han sido ejecutados. En nuestro casos el equipo informático empleado ha sido un ordenador con un procesador Intel Core i7-2600 CPU 3.4 Ghz running cuyo sistema operativo es Microsoft Windows 7.

La distribución de los resultados en las distintas tablas viene dada de la siguiente manera, en las Tablas 3.1, 3.2 y 3.3 las primeras tres columnas representan el número de localizaciones, n, la capacidad máxima de carga del vehículo, Q, y la semilla que hemos tomado. Concretamente hemos tomado cuatro semillas distintas, las cuales han sido dadas por cuatro archivos del tipo *nombre.tsp* en los que se encuentran almacenados los datos (coordenadas, distancias y demandas de las distintas localizaciones) que son necesarios proporcionarle al programa para que resuelva de forma efectiva el modelo, y mediante las cuales se cumple la aleatoriedad de los datos de entradas. Las siguientes cinco columnas de dichas tablas se corresponden con los diferentes métodos de resolución:

- M_{CF} **:** corresponde al modelo carga-flujo (2.7)–(2.13) que es resuelto tal cual está definido por XPRESS.
- M_{Mul} **:** correspondiente al modelo con variable multiflujo (2.14)–(2.24) que es resuelto por el programa ya indicado.
- M_{Pot} **:** correspondiente al modelo con variable potencial (2.25)–(2.30) que es resuelto por el programa ya indicado.
- M_{Cap} **:**correspondiente al modelo con restricciones de capacidad (2.31)–(2.34) al cual denominamos *1-PDTSP capacity* que es resuelto por el programa ya indicado, y cuya manera de proceder es resolver el modelo sin las restricciones (2.33) y una vez resuelto se verifica si hay alguna restricción violada y en caso de haberla se vuelve a resolver el modelo con dicha restricción añadida..
- M_{Cortes} **:**correspondiente al modelo con restricciones de capacidad (2.31)–(2.34) que es resuelto con el algoritmo de Ramificación y Corte, con el cual se verifica primero si hay alguna restricción (2.33) violada sin necesidad de esperar a que la solución del problema sea entera, por el programa ya indicado.

Además, las Tablas 3.2 y 3.3 poseen una columna adicional en la que se reflejan los tiempos computacionales resultantes de la implimentación de un código que ha sido diseñado y creado de forma particular y concreta en *C++* para la resolución del *problema del viajante de comercio con recogida y entrega de mercancía*, los cuales hemos obtenido de la Tesis Doctoral del profesor Hipólito Hernández Pérez [2]. Para la ejecución de dicho programa fue empleado un ordenador con un procesador AMD Athlon XP 2600+ (2.08 GHz.). Posteriormente, la Tabla 3.4 mostrará los promedios totales de los tiempos computacionales de

cada modelo para cada una de las cantidades correspondientes al número de localizaciones. Por último, señalamos que las cantidades correspondientes al número de localizaciones y las capacidades máximas de carga del vehículo que hemos tomado son n en $\{20, 30, 40\}$ y Q en $\{10, 15, 20, 25\}$, respectivamente, y además, para los tiempos computacionales se ha fijado un tiempo límite de 1 hora.

Procedemos ahora a mostrar cada una de las tablas de los resultados y explicar el comportamientos que se refleja en cada una de ellas.

En primer lugar, tenemos la Tabla 3.1 en la que se representan los tiempos computacionales obtenidos para $n = 20$ y para cada una de las semillas, respecto al aumento de la capacidad máxima de carga del vehículo.

n	Q	Semilla	M$_{CF}$	M$_{Mul}$	M$_{Pot}$	M$_{Cap}$	M$_{Cortes}$
20	10	n20A.tsp	3,0	22,5	5,5	14,7	14,0
		n20B.tsp	3,9	14,9	2,4	6,5	4,8
		n20C.tsp	2,6	17,0	4,3	11,5	6,0
		n20D.tsp	1,9	12,6	2,8	6,2	0,6
			2,8	16,8	3,7	9,7	6,3
	15	n20A.tsp	1,4	6,0	2,1	0,3	0,3
		n20B.tsp	1,8	8,8	1,7	0,0	0,0
		n20C.tsp	2,6	14,4	4,9	4,5	4,8
		n20D.tsp	2,2	5,6	1,3	0,9	0,4
			2,0	8,7	2,5	1,4	1,4
	20	n20A.tsp	0,3	0,5	1,3	0,0	0,0
		n20B.tsp	1,5	4,0	1,6	0,0	0,0
		n20C.tsp	2,4	11,7	2,4	0,5	0,2
		n20D.tsp	2,8	14,3	2,8	2,0	1,9
			1,7	7,6	2,1	0,6	0,5
	25	n20A.tsp	1,2	0,5	1,1	0,1	0,0
		n20B.tsp	0,8	0,5	1,3	0,0	0,0
		n20C.tsp	1,3	2,7	0,6	0,1	0,0
		n20D.tsp	1,8	6,0	2,2	0,1	0,4
			1,3	2,4	1,3	0,1	0,1
Total 20			2,0	8,9	2,4	3,0	2,1

Tabla 3.1. Resultados computacionales para n=20

Lo primero que podemos observar es que en cuanto aumenta la capacidad máxima de carga del vehículo, Q, los tiempos computacionales disminuyen de forma continua en cada modelo respecto a cada una de las semillas. También vemos que el mejor modelo para ejecutar y resolver el problema es el modelo M_{CF}, aunque se observe que el modelo M_{Cortes} funcione y proceda igual de bien.

Por último, vemos claramente que el peor modelo es el modelo M_{Mul} puesto que es el posee los tiempos más largos.

En segundo lugar, tenemos la Tabla 3.2 en la que se representan los tiempos computacionales obtenidos para $n = 30$ y para cada una de las semillas, respecto al aumento de la capacidad máxima de carga del vehículo.

n	Q	Semilla	M_{CF}	M_{Mul}	M_{Pot}	M_{Cap}	M_{Cortes}	[2]
30	10	n30A.tsp	284,9	2338,3	42,7	198,9	455,9	1,0
		n30B.tsp	7,2	341,8	4,8	99,8	16,5	0,1
		n30C.tsp	609,8	3600,0	413,9	135,1	3223,3	0,4
		n30D.tsp	169,4	607,6	30,7	105,4	126,3	0,3
			267,8	1722,0	123,0	134,8	955,5	0,5
	15	n30A.tsp	148,0	3600,0	39,2	46,9	161,0	0,3
		n30B.tsp	643,8	3600,0	114,8	49,4	102,2	0,2
		n30C.tsp	10,3	61,9	7,6	3,1	1,3	0,1
		n30D.tsp	63,0	244,8	37,1	31,8	44,9	0,3
			216,3	1876,5	49,7	32,8	77,4	0,2
	20	n30A.tsp	7,4	55,9	11,3	6,2	11,1	0,1
		n30B.tsp	74,2	836,1	33,2	8,3	18,4	0,1
		n30C.tsp	30,1	570,7	16,5	2,8	7,7	0,1
		n30D.tsp	34,7	233,9	56,8	18,1	41,2	0,2
			36,6	424,1	29,4	8,9	19,6	0,1
	25	n30A.tsp	2,1	22,5	5,0	0,5	0,1	0,0
		n30B.tsp	18,0	92,6	14,8	2,7	4,4	0,0
		n30C.tsp	5,5	71,3	7,4	1,0	0,2	0,0
		n30D.tsp	9,0	135,7	9,3	3,3	1,8	0,1
			8,7	80,5	9,1	1,9	1,6	0,0
Total 30			132,3	1025,8	52,8	44,6	263,5	0,2

Tabla 3.2. Resultados computacionales para n=30

Observamos que al contrario de como lo hacían en la Tabla 3.1, en este segunda tabla se tiene que al aumentar la capacidad máxima de carga del vehículo los tiempos computacionales no disminuyen de forma continua puesto que vemos que para la semilla *" n30B.tsp "* los mayores tiempos computacionales, exceptuando los dados por el modelo M_{Cap}, no se obtiene para la capacidad Q más pequeña sino que se han obtenido para $Q = 15$. Este hecho se ve reflejado de un modo más notable en el modelo M_{Mul}, lo cual ayuda a que volvamos a tener que este modelo es el que posea mayores tiempos computacionales, y sea así el modelo menos efectivo. Posteriormente, vemos que el mejor modelo para ejecutar y resolver el problema es el modelo M_{Cap} pues es el modelo que mejores tiempos computacionales posee.

En tercer lugar, tenemos la Tabla 3.3 en la que se representan los tiempos computacionales obtenidos para $n = 40$ y para cada una de las semillas, respecto al aumento de la capacidad máxima de carga del vehículo.

n	Q	Semilla	M_{CF}	M_{Mul}	M_{Pot}	M_{Cap}	M_{Cortes}	[2]
40	10	n40A.tsp	3600,0	3600,0	3600,0	2262,0	3600,0	5,2
		n40B.tsp	3600,0	3600,0	3600,0	3491,7	3600,0	16,2
		n40C.tsp	3310,0	3600,0	1348,9	1299,4	548,9	0,3
		n40D.tsp	3600,0	3600,0	3600,0	3600,0	3600,0	7,3
			3527,1	3600,0	3037,0	2663,2	2841,6	7,2
	15	n40A.tsp	3600,0	3600,3	2941,5	115,8	714,5	0,9
		n40B.tsp	21,5	195,0	36,4	16,4	20,7	0,1
		n40C.tsp	3600,0	3600,0	3600,0	1141,2	21,0	0,4
		n40D.tsp	1207,7	3600,0	363,4	620,5	141,2	0,3
			2107,2	2748,9	1735,2	473,4	224,4	0,4
	20	n40A.tsp	275,3	3600,0	349,9	50,8	34,5	0,4
		n40B.tsp	10,8	188,0	8,2	3,7	12,9	0,1
		n40C.tsp	3600,0	3600,0	3600,0	171,7	1015,9	0,5
		n40D.tsp	1171,4	3600,0	227,4	100,9	170,2	1,0
			1264,3	2747,2	1046,3	81,8	308,4	0,5
	25	n40A.tsp	9,3	138,7	8,6	1,2	0,5	0,0
		n40B.tsp	7,4	68,8	9,1	2,4	6,7	0,1
		n40C.tsp	625,6	3600,0	1111,5	67,3	206,4	0,3
		n40D.tsp	57,2	3214,7	403,5	1,1	0,3	0,1
			174,9	1755,8	383,2	18,0	53,5	0,1
Total 40			1768,4	2712,9	1550,4	809,1	857,0	1,8

Tabla 3.3. Resultados computacionales para n=40

En esta otra tabla, el modelo que posee menores tiempos computacionales y por tanto el método más efectivo es el modelo M_{Cap}, aunque observamos que el modelo M_{Cortes} proporcione tiempos igual de buenos. Nuevamente observamos alteraciones en los tiempos computacionales, pero en este caso respecto a las semillas *" n30A.tsp "*, *" n30C.tsp "* y *" n30D.tsp "* para las cuales vemos que los mayores tiempos computacionales se han obtenido en $Q = 15$ y en $Q = 20$ menos en el modelo M_{Mul} en el que se obtuvo para $Q = 25$, respectivamente. Además, vemos que para el modelo M_{Cortes} este suceso solo tiene lugar para la semilla *" n30C.tsp "* pues para las demás semillas los tiempos disminuyen sin alteraciones al aumentar la capacidad. Por último vemos que el modelo M_{Mul} vuelve a tener los mayores tiempos computacionales.

Como ya enunciamos en la introducción de esta sección, en la Tablas 3.2 y 3.3 se han añadido una columna en la cual hemos colocado los tiempos compu-

tacionales resultantes de la implimentación de un código diseñado y creado de forma particular y concreta en *C++* para la resolución del *problema del viajante de comercio con recogida y entrega de mercancía*. Fijándonos en dicha columna en ambas tablas podemos ver que los tiempos computacionales dados por dicho código son claramente mejores que los resultantes de nuestros cinco modelos, lo cual es lógico debido a que el mencionado código ha sido programado de forma particular y detallada para el *problema del viajante de comercio con recogida y entrega de mercancía*.

Por último, tenemos la Tabla 3.4 en la que como hemos dicho anteriormente mostramos los promedios de los tiempos computacionales correspondientes a cada modelo para $n = 20, 30, 40$ y el promedio total obtenido de cada modelo:

n	M_{CF}	M_{Mul}	M_{Pot}	M_{Cap}	M_{Cortes}
20	2,0	8,9	2,4	3,0	2,1
30	132,3	1025,8	52,8	44,6	263,5
40	1768,4	2712,9	1550,4	809,1	857,0
Promedio Total	634,2	1249,2	535,2	285,6	374,2

Tabla 3.4. Promedios totales

Finalmente, en esta última tabla, vemos de forma evidente que con el aumento del número de localizaciones, n, el modelo que mejores tiempos computacionales posee y por tanto el más efectivo para nuestro problema es el modelo M_{Cap}, seguido por poco del modelo M_{Cortes}, y que el modelo M_{Mul} es el menos efectivo pues posee los mayores tiempos computacionales.

4

Conclusiones

Para la ejecución del presente trabajo se han realizado un estudio previo de trabajos sobre el *problema de viajante de comercio (TSP)* y más concretamente sobre el *problema de viajante de comercio con recogida y entrega de mercancía (1-PDTSP)*, como son los realizados por los profesores Hipólito Hernández Pérez [2] y Juan José Salazar González [1]. Durante el estudio previo se han encontrado además algunos artículos relacionados con el análisis de estos problemas.

Una vez finalizado dicho estudio, se hizo una selección en la que terminamos optando por los cinco modelos con los cuales se ha efectuado este trabajo de final de carrera, que son como ya sabemos: el Modelo 1-PDTSP con restricciones de subciclos, el Modelo con variables flujo, el Modelo 1-PDTSP con variables multiflujo, el Modelo 1-PDTSP con variable de potencial y el Modelo 1-PDTSP de Ramificación y Corte. Luego procedimos a la programación en el Xpress-Mosel de tales modelos, la cual no fue nada sencilla en el caso de la programación del primer y último modelo citado anteriormente debido a la complejidad a la hora de programar la función *" break_capacity_and_subtour "*, que era la encargada de buscar si hay restricciones de capacidad y evitar que se formen subciclos, en el caso del modelo con restricciones de subciclos, y en el caso de el modelo de Ramificación y Corte debido a la complejidad al programar las funciones *" cb_entera "* y *" cb_fraccional "*, que fueron las encargadas de verificar que la solución obtenida es entera mediante la ejecución de la correspondiente llamada y de resolver el problema de separación cuando se esté en un nodo del árbol de ramificación, respectivamente.

Cabe destacar que, a pesar de que en la implementación del modelo de Ramificación y Corte, el empleo del comando *setcallback*, el cual hemos explicado en la sección 3.1 del capítulo anterior, haya reducido de forma considerable los tiempos computacionales para la resolucion del 1-PDTSP, hemos obtenido finalmente que el modelo más efectivo no ha sido este sino el modelo 1-PDTSP con restricciones de subciclos. Debido al hecho de que el método de Ramificación

y Corte emplea menor número de restricciones, lo que provoca que sus tiempos computacionales deberían de ser teóricamente menores que los dados por cualquier otro modelo, entonces nos hace concluir que este hecho puede ser un buen punto de partida para posibles futuros proyectos.

A

Apéndice I

Debido a que llegados a este punto de la presente memoria disponemos de poco margen de páginas para poder mostrar todos los modelos programados en Xpress-Mosel, a continuación mostraremos el código programado correspondiente al modelo 1-PDTSP resuelto mediante el algoritmo de Ramificación y Corte puesto que ha sido el modelo que mayor importancia ha tenido en el transcurso de este trabajo dada la dedicación que ha llevado su estudio y su posterior programación, y el procedimiento llamado *" break_capacity_and_subtour"* perteneciente al modelo 1-PDTSP capacity y que es la encargada de evitar que se produzcan subciclos en la solución del problema.

A.1. Modelo 1-PDTSP de ramificación y corte

```
(!*********************************************************************
TSP: Resolvemos el TSP, a partir del metodo exacto de resolucion
llamado ramificacion y corte. Para encontrar los cortes se utiliza el
procedimiento de hallar el flujo max (o corte min).
Se hace uso de funciones de llamada recursiva mediante la
sentencia "setcallback".
*********************************************************************
!Se cargan las librerias y se le da nombre al modelo
!-------------------------------------------------------------------
model "Tsp subtour elimination"
uses "mmxprs","mmive", "mmsystem"

parameters
File="n20q10A.tsp"
OUTFILE="resultados.txt"
NESTACIONES=20
```

```
CAPACIDAD = 10
end-parameters


!Se modifican los parametros utilizados por el resolutor
!-------------------------------------------------------------------------
setparam("XPRS_HEURSTRATEGY", 0)  !Estrategia heuristica, 0 indica
no heuristico
setparam("XPRS_CUTSTRATEGY", 0)  !Estrategia de corte, 0 no incluye
cortes de criterio propio
setparam("XPRS_PRESOLVE", 0)  !Preresolutor, 0 no se aplica el presolve
setparam("XPRS_EXTRAROWS", 5000)  !Reserva columnas extra de la matriz


!Se advierte de la existencia de determinadas funciones y procedimientos
que se encuentran en adelante
!-------------------------------------------------------------------------
forward public function cb_fraccional:boolean
forward public procedure cb_entera(isheur:boolean,cutoff:real)
forward procedure fichero_datos
forward procedure write_resultados
forward procedure draw

declarations
MARGEN = 1
MAXCOORD = 500

root    = 1
ESTACIONES  = 1..NESTACIONES
epsilon = 0.0001

temp,cut_id: integer
K: real
x: array(ESTACIONES) of real
y: array(ESTACIONES) of real
DIST:  array(ESTACIONES,ESTACIONES) of integer
DEMANDA: array(ESTACIONES)of integer !Cantidad de carga a recoger/dejar
! ASSIG: array(ESTACIONES,ESTACIONES)of mpvar ! 1 si vamos de la
localizacion i hasta la j
addcut_in_cb_entera: boolean  ! verdadero o falso
cut: linctr
assig: array(ESTACIONES,ESTACIONES) of mpvar
end-declarations
```

```
addcut_in_cb_entera:= false


!Se establecen los parametros leyendo de fichero
!-------------------------------------------------------------------

!Lee el fichero de datos
fichero_datos

!Calculamos las distancias
forall(i,j in ESTACIONES | i<j) DIST(i,j):=round(sqrt( (x(i)-x(j))*
(x(i)-x(j))+(y(i)-y(j))*(y(i)-y(j)) ))
forall(i,j in ESTACIONES | i<j) DIST(j,i):=DIST(i,j)

K := sum(i in ESTACIONES | DEMANDA(i) > 0) (DEMANDA(i)) / CAPACIDAD

! MODELO
!===================================================================

writeln("Resolvemos el TSP con n=", NESTACIONES," ciudades.")

!Se establece el modelo
!-------------------------------------------------------------------
TotalDist:= sum(i,j in ESTACIONES | i<>j) DIST(i,j)*assig(i,j)
!Sujeto a:
forall(i in ESTACIONES) sum(j in ESTACIONES | i<>j) assig(j,i) = 1
!Entramos a cada ciudad una sola vez
forall(i in ESTACIONES) sum(j in ESTACIONES | i<>j) assig(i,j) = 1
!Salimos de cada ciudad una sola vez
forall(i,j in ESTACIONES | i<j) assig(i,j)+assig(j,i) <= 1
!Elimina los subciclos de tamaÃ±o 2
forall(i,j in ESTACIONES | i<>j and abs(DEMANDA(i) + DEMANDA(j)) > CAPACIDAD)
 assig(i,j) = 0 ! Fija a 0 aarcos no factibles
forall(i,j in ESTACIONES | i<>j) assig(i,j) is_binary
!La variable la forzamos a ser binaria

!Se genera el procedimiento de ramificacion y corte, mediante
!funciones recursivas
!-------------------------------------------------------------------
setcallback(XPRS_CB_CUTMGR, "cb_fraccional")
setcallback(XPRS_CB_PREINTSOL, "cb_entera")
```

```
cut_id := 0

now1:= datetime(SYS_NOW) !Tiempo de inicio
mips := 1;
minimize(TotalDist)

now2:= datetime(SYS_NOW) !Tiempo final

!Escribimos en el fichero de resultados:
fopen(OUTFILE,F_OUTPUT+F_APPEND)
write_resultados
fclose(F_OUTPUT)

writeln("Tiempo", now2-now1)
writeln(" Cost: ", getobjval)
draw

! FUNCIONES Y PROCEMIENTOS
!=====================================================================

!Procedimiento llamado cuando es encontrada una solucion entera
mediante aproximaciones o ramificacion y acotacion.
!Si la solucion tiene subciclos es desechada, siendo no valida
la cota superior asociada.
!---------------------------------------------------------------------
public procedure cb_entera(isheur:boolean,cutoff:real)
declarations
TOUR,SMALLEST,ALLESTACIONES: set of integer
node: integer
constraints: integer
end-declarations

!writeln("CB_ENTERA.  isheur= ", isheur, "  cutoff= ", cutoff )

draw

!Para cada nodo se indentifica cual es su hijo
forall(i in ESTACIONES)
NEXTC(i):= ceil( getsol( sum(j in ESTACIONES) j*assig(i,j))-epsilon)

!Este bucle busca restricciones de capacidad
ALLESTACIONES:={}
constraints:= 0
```

```
forall(k in ESTACIONES) do
if(k not in ALLESTACIONES) then
TOUR:={}
SMALLEST:={}
first:=k
cap:= 0
index:=0
maxcap:= 0
max_index:= 0
mincap:= 0
min_index:= 0
repeat
TOUR+={first}
cap += DEMANDA(first)
index += 1
if(cap >=  maxcap) then
maxcap:= cap
max_index:= index
elif(cap <= mincap) then
mincap:= cap
min_index:= index
end-if
if(maxcap - mincap > CAPACIDAD) then
! Hemos encontrado una restriccion de capacidad violada
indice:=0;
! truco para ver desde donde empezamos a contar el conjunto
forall(j in TOUR) do
if (indice >= max_index or indice >= min_index) then
SMALLEST+={j}
end-if
indice += 1
end-do

constraints += 1
cut_id += 1

size:= getsize(SMALLEST)

!write("Nodo:" , getparam("XPRS_NODES"), "  profundidad= ",
getparam("XPRS_NODEDEPTH"))
!write("  Costo: ", getparam("XPRS_LPOBJVAL"), "  Corte", cut_id,
" Cap: rhs=", size - 2,"{")
!forall(i in SMALLEST) write(i,",")
```

```
!writeln("}")
!Agregamos el corte, si procede
cut:=sum(i,j in SMALLEST | i<>j ) assig(i,j) - size + 2
addcut(cut_id, CT_LEQ, cut)
! Hacemos este truco para reiniciar el recuento en el tour
SMALLEST:={}
cap:= DEMANDA(first)
maxcap:= 0
mincap:= 0
max_index:= index-1
min_index:= index-1
if(cap >=  maxcap) then
maxcap:= cap
max_index:= index
elif(cap <= mincap) then
mincap:= cap
min_index:= index
end-if
end-if
first:=NEXTC(first)
until first=k
end-if
ALLESTACIONES+=TOUR
end-do

size := getsize(TOUR)

!Si hay subciclos identifica el menor conjunto de nodos generados de
uno de ellos
if constraints = 0 and size < NESTACIONES then
ALLESTACIONES:={}

forall(k in ESTACIONES) do
if (addcut_in_cb_entera) then
TOUR:={}
first:=k
repeat
TOUR+={first}
first:=NEXTC(first)
until first=k
!insertamos las restricciones de subciclo
constraints += 1
cut_id += 1
```

```
size:= getsize(TOUR)
!sum(i,j in TOUR | i<>j ) assig(i,j) <= size - 1
!write("Nodo:" , getparam("XPRS_NODES"), "profundidad= ",
getparam("XPRS_NODEDEPTH"))
!write("  Costo: ", getparam("XPRS_LPOBJVAL"), "  Corte", cut_id,
"SEC: rhs=", size - 1," {")
!forall(i in TOUR) write(i,",")
!writeln("}")

!Agrega el corte, si procede
cut:=sum(i,j in TOUR | i<>j ) assig(i,j) - size + 1
addcut(cut_id, CT_LEQ, cut)
end-if
end-do
ALLESTACIONES+=TOUR
end-if

!IVEpause(""+mips)

if(constraints > 0) then
rejectintsol
end-if

end-procedure


!Subrutina que identifica cortes violados, mediante un problema de
!flujo max. Este genera una restriccion que es agregada
!al nodo actual y a sus hijos. Se procede a ramificar cuando
!no sean encontrados flujos max menores que 1, desde la raiz
!al resto.
!-------------------------------------------------------------------------
public function cb_fraccional:boolean
declarations
ESTACIONES_PLUS  = 1..(NESTACIONES+2)
Capacity,Flow:array (ESTACIONES_PLUS,ESTACIONES_PLUS) of real
cap, KS: real
first,last, SINK,SOURCE :integer
TOUR  : array(ESTACIONES_PLUS) of integer
inside: array(ESTACIONES_PLUS) of boolean
maxflow,maxflow2: mpproblem
amount:real
flow: array(ESTACIONES_PLUS,ESTACIONES_PLUS) of mpvar
```

```
end-declarations

addcut_in_cb_entera:= true

draw

!Creamos un grafo con cargas
forall(n,m in ESTACIONES|n<>m) Capacity(n,m) := getsol(assig(n,m))
forall(i,j in ESTACIONES|i<>j and Capacity(i,j)<epsilon )
Capacity(i,j) := 0

cap:= CAPACIDAD
forall(i in ESTACIONES | DEMANDA(i) > 0) Capacity(NESTACIONES+1,i)
:= DEMANDA(i) / cap
!forall(i in ESTACIONES | DEMANDA(i) <= 0) Capacity(NESTACIONES+1,i)
:= 0
forall(i in ESTACIONES | DEMANDA(i) < 0) Capacity(i,NESTACIONES+2)
:= -DEMANDA(i) / cap
!forall(i in ESTACIONES | DEMANDA(i) >= 0) Capacity(i,NESTACIONES+2)
:= 0

returned:=false

with maxflow2 do
reset( maxflow2 )
forall(n,m in ESTACIONES_PLUS | n<>m and Capacity(n,m)>epsilon )
create(flow(n,m))
forall(n in ESTACIONES)
sum(m in ESTACIONES_PLUS | m<>n and exists(flow(m,n))) flow(m,n) =
sum(m in ESTACIONES_PLUS|m<>n and exists(flow(n,m))) flow(n,m)
! forall(n in ESTACIONES_PLUS | n<> NESTACIONES+1 and exists(flow
(n, NESTACIONES+1)))
flow(n,NESTACIONES+1) = 0
forall(n,m in ESTACIONES_PLUS | n<>m and exists(flow(n,m)) )  flow(n,m)
<= Capacity(n,m)

! sum(n,m in 1..(NESTACIONES+2)|n<>m) assig(n,m)>=ceil(abs(K))

maximize( sum(n in ESTACIONES_PLUS | n<> NESTACIONES+1 and exists
(flow(NESTACIONES+1,n)))flow(NESTACIONES+1,n) )

amount := getobjval
forall (n,m in ESTACIONES_PLUS | n<>m) Flow (n,m):= getsol(flow(n,m))
```

```
end-do

if amount < K - epsilon then
forall(i in ESTACIONES) inside(i):=false
TOUR(1):= NESTACIONES+1
inside(NESTACIONES+1):= TRUE
last :=1
while (last>0) do
u := TOUR(last)
last := last-1
forall ( v in ESTACIONES_PLUS | v<>u ) do
if ( Capacity(u,v) - Flow(u,v) > epsilon or Flow(v,u) > epsilon ) and
(not inside(v)) then
last := last+1
TOUR(last):= v
inside(v):=true
end-if
end-do
end-do

!Agregamos el corte oportuno
KS :=  ceil(abs(sum(i in ESTACIONES | inside(i)=true) DEMANDA(i)) / cap)
cut  := ( sum(i,k in ESTACIONES | inside(i)=true and inside(k)=false)
assig(i,k))-KS

addcut(1, CT_GEQ, cut)
size := 0
forall(i in ESTACIONES | inside(i)=true) do
size += 1
end-do
!writeln("Nodo:", getparam("XPRS_NODES"), "  profundidad= ",getparam
("XPRS_NODEDEPTH")," Costo: ", getparam("XPRS_LPOBJVAL"))
!write("  Corte:  ", cut_id, "  Cap : rhs= ", size - KS," {")
! forall(i in ESTACIONES | inside(i)=true) do
!write(i,",")
!end-do
!writeln("}")

returned:=true
end-if

if(returned = false) then
```

```
!Resolvemos el problema de flujo max
SOURCE:=root
forall(j in ESTACIONES|j<>SOURCE)do
SINK:=j
with maxflow do
reset( maxflow )
forall(n,m in ESTACIONES |n<>m and Capacity(n,m)>epsilon )
create(flow(n,m))
forall(n in ESTACIONES | n<>SOURCE and n<>SINK)
sum(m in ESTACIONES|m<>n and exists(flow(m,n))) flow(m,n) =
sum(m in ESTACIONES|m<>n and exists(flow(n,m))) flow(n,m)
forall(n in ESTACIONES | n<>SOURCE and exists(flow(n,SOURCE)) )
flow(n,SOURCE) = 0
forall(n,m in ESTACIONES | n<>m and exists(flow(n,m)) ) flow(n,m)
<=Capacity(n,m)

maximize( sum(n in ESTACIONES|n<>SOURCE and exists(flow(SOURCE,n)))
flow(SOURCE,n) )

amount := getobjval
forall (n,m in ESTACIONES | n<>m) Flow (n,m):= getsol(flow(n,m))
end-do

!Identificamos si el flujo es menor que uno, para posteriormente encontrar
una restriccion de subciclos violada
  if amount < 1-epsilon then
forall(i in ESTACIONES) inside(i):=false
TOUR(1):= SOURCE
inside(SOURCE):= TRUE
last :=1
while (last>0) do
u := TOUR(last)
last := last-1
forall ( v in ESTACIONES | v<>u ) do
if ( Capacity(u,v) - Flow(u,v) > epsilon or Flow(v,u) > epsilon )
and (not inside(v)) then
last := last+1
TOUR(last):= v
inside(v):=true
end-if
end-do
end-do
```

```
cut_id += 1
!Agragamos el corte oportuno
cut  := ( sum(i,k in ESTACIONES | inside(i)=true and inside(k)=false)
assig(i,k) ) -1
addcut(cut_id, CT_GEQ, cut)
size := 0
forall(i in ESTACIONES | inside(i)=true) do
size += 1
end-do
!writeln("Nodo:", getparam("XPRS_NODES"), "  profundidad= ",getparam
("XPRS_NODEDEPTH")," Costo: ", getparam("XPRS_LPOBJVAL"))
!write("  Corte:  ", cut_id, "  SEC: rhs= ", size - 1," {")
!forall(i in ESTACIONES | inside(i)=true) do
! write(i,",")
!end-do
!writeln("}")

returned:=true
break
end-if
end-do
end-if

if returned = false then
writeln("Ramificando")
end-if
end-function


! Procedimiento de lectura del fichero
!-------------------------------------------------------------------------

procedure fichero_datos

fopen(File,F_INPUT)
readln !readln(NCITIES)
readln
readln
readln
readln
readln
forall (i in 1..NESTACIONES) readln(temp, x(i), y(i))
readln
```

```
forall (i in 1..NESTACIONES) readln
readln
forall (i in 1..NESTACIONES) readln (temp, DEMANDA(i))
fclose(F_INPUT)

end-procedure

! Procedimiento de escritura
!-------------------------------------------------------------------
procedure write_resultados
writeln(File, "\t", "M_Cortes\t", NESTACIONES, "\t", CAPACIDAD , "\t",
getobjval, "\t", now2-now1)
end-procedure

!Procedimiento para dibujar el grafo
!-------------------------------------------------------------------

procedure draw
IVEerase
!IVEzoom(-MAXCOORD-MARGEN,-MAXCOORD-MARGEN,MAXCOORD+MARGEN,
MAXCOORD+MARGEN)
points:=IVEaddplot("EstaciÃ³n",IVE_RED)
!roads:=IVEaddplot("Camino",IVE_BLUE)
qi:=IVEaddplot("Carga",IVE_CYAN)
q:=IVEaddplot("Demanda",IVE_GREEN)
roads1:=IVEaddplot("0<cables<=0.25",IVE_BLUE)
roads2:=IVEaddplot("0.25<cables<=0.5",IVE_YELLOW)
roads3:=IVEaddplot("0.5<cables<=0.75",IVE_GREEN)
roads4:=IVEaddplot("0.75<cables<=1",IVE_BLACK)

forall(i in 1..NESTACIONES) do
IVEdrawlabel(points,x(i),y(i),""+i)
IVEdrawlabel(q,x(i)+0.04*MAXCOORD,y(i)+0.04*MAXCOORD,""+DEMANDA(i))
end-do
!draw links

forall(i,j in 1..NESTACIONES | getsol(assig(i,j))>epsilon) do
if getsol(assig(i,j)) < 0.25+epsilon then
IVEdrawarrow(roads1,x(i),y(i),x(j),y(j))
else
if getsol(assig(i,j)) <0.5+epsilon then
IVEdrawarrow(roads2,x(i),y(i),x(j),y(j))
else
```

```
if getsol(assig(i,j)) < 0.75+epsilon then
IVEdrawarrow(roads3,x(i),y(i),x(j),y(j))
else
IVEdrawarrow(roads4,x(i),y(i),x(j),y(j))
end-if
end-if
end-if
end-do
end-procedure

end-model
```

A.2. Procedimiento

```
procedure break_capacity_and_subtour
declarations
cap, maxcap, mincap, max_index, min_index: integer
TOUR,SMALLEST,ALLCITIES: set of integer
constraints: integer
end-declarations

! Dibujamos los resultados hasta ahora
draw


! Consigue la siguiente ciudad
forall(i in ESTACIONES) do
NEXTC(i):= integer(round(getsol(sum(j in ESTACIONES) j*ASSIG(i,j) )))
end-do


! Obtener (sub)tour que contiene la ciudad 1

! Bucle que se encarga de buscar las restricciones de capacidad
ALLCITIES:={}
constraints:= 0
forall(k in ESTACIONES) do
if(k not in ALLCITIES) then
TOUR:={}
SMALLEST:={}
first:=k
cap:= 0
```

```
index:=0
maxcap:= 0
max_index:= 0
mincap:= 0
min_index:= 0
repeat
TOUR+={first}
cap += DEMANDA(first)
index += 1
if(cap >=  maxcap) then
maxcap:= cap
max_index:= index
elif(cap <= mincap) then
mincap:= cap
min_index:= index
end-if
if(maxcap - mincap > CAPACIDAD) then
! Hemos encontrado una de las restricciones que ha sido violada
indice:=0;
! truco para ver desde donde empezamos a contar el conjunto
forall(j in TOUR) do
if (indice >= max_index or indice >= min_index) then
SMALLEST+={j}
end-if
indice += 1
end-do
constraints += 1
size:= getsize(SMALLEST)
! Insertamos las restricciones de capacidad
sum(i,j in SMALLEST | i<>j ) ASSIG(i,j) <= size - 2
! write(mips,") Cost: ", getobjval,"   ;  rhs=",size-2," ={")
! forall(i in SMALLEST) write(i,",")
! writeln("}")
! Hacemos este truco para reiniciar el recuento en el tour
SMALLEST:={}
cap:= DEMANDA(first)
maxcap:= 0
mincap:= 0
max_index:= index-1
min_index:= index-1
if(cap >=  maxcap) then
maxcap:= cap
max_index:= index
```

```
elif(cap <= mincap) then
mincap:= cap
min_index:= index
end-if
end-if
first:=NEXTC(first)
until first=k
ALLCITIES+=TOUR
end-if
end-do

size := getsize(TOUR)
! Si no hemos encontrado restricciones de capacidad y hay mas de un (sub)tour
! es decir que se producen subciclos los insertamos
if constraints = 0 and size < NESTACIONES then
ALLCITIES:={}
forall(k in ESTACIONES) do
if(k not in ALLCITIES) then
TOUR:={}
first:=k
repeat
TOUR+={first}
first:=NEXTC(first)
until first=k
!insertamos las restricciones de subciclo
constraints += 1
size:= getsize(TOUR)
sum(i,j in TOUR | i<>j ) ASSIG(i,j) <= size - 1
!write(mips,") Cost: ", getobjval,"   ;  rhs=",size - 1," {")
!forall(i in TOUR) write(i,",")
!writeln("}")
end-if
ALLCITIES+=TOUR
end-do
end-if

mips := mips+1
!IVEpause(""+mips)
if(constraints > 0) then
! Re-solve the problem
minimize(TotalCost)
! Recurse
break_capacity_and_subtour
```

```
end-if
end-procedure
```

Bibliografía

[1] Juan José Salazar González, *Programación Matemática*, Ed. Diaz de Santos, 2001.

[2] Hipólito Hernández Pérez *Procedimientos exactos y heurísticos para resolver problemas de rutas con recogida y entrega de mercancía* Serie Tesis Doctorales, 2004/05.

[3] Gregory Gutin, Abraham P. Punnen *The Traveling Salesman Problem and its variations* Ed. Dordrecht, Kluwer Academic, cop. 2002.

[4] David Bloomberg, Stephen LeMay y Joe B. Hanna *Logistics* 2002.

[5] M. L. Stockdale *El problema del viajante: un algoritmo heurístico y una aplicación.El problema del viajante: un algoritmo heurístico y una aplicación.* PhD thesis, 2011.

[6] FICO: XPress-Mosel (User Guide, Release 4.8), October 2017

[7] G. Dantzig, R. Fulkerson S. Johnson *Solution of a Large-Scale Travelling-Salesman Problem* Journal of the Operations Research Society of America, Vol. 2, Nº. 4, pp. 393-410, November 1954.

[8] M. Held and R. M. Karp *The Travelling Salesman Problem and Minimum Spanning Trees: part II* Mathematical Programming, Vol. 1, pp. 6-25, 1971.

[9] P. M. Camerini, L. Fratta and F. Maffioli *On Improving Relaxation Methods by Modified Techniques* Mathematical Programming, Vol. 3, pp. 26-34, 1975.

[10] M. Grötschel *Polyedrische Charakterisierunge Kombinatorischer Optimierungsprobleme* Hain, Meisenheim am Glan, 1977.

[11] H. Crowder and M. W. Padberg *Solving large scale symmetric travelling salesman problems to optimality* In Management, Vol. 26, pp. 485-509, 1980.

[12] M. Padberg and Rinaldi *Optimization of a 532-city symmetric traveling salesman problem by branch and cut* In Operations Research Letters, Vol. 6, No 1, pp. 1-7, 1987.

[13] M. Grötschel O. Holland *A Cutting Plane Algorithm for Minimum Perfect 2-Matchings* Computing, Vol. 39, pp. 327-344, 1987.

[14] M. Padberg and G. Rinaldi *A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Travelling Salesman Problems* SIAM Review, Vol. 33, Nº. 1, pp. 60-100, March 1991.

[15] D. Apllegate, R. Bixby, V. Chvátal and W. Cook *Finding cuts in the TSP (A preliminary report)* DIMACS Technical Report, 95-05, March 1995

[16] W. Cook, D. Espinoza and M. Goycoolea *Integer Programming and Combinatorial Optimization, Proceedings* Lecture Notes in Computer Science, Vol. 3509, pp. 452-467, 2005.

[17] David L. Applegate, Robert E. Bixby, Vasek Chvátal William J. Cook *The Travelling Salesman Problem, A Computational Study* Princeton University Press, 2006.

Pick-up and Delivery Travelling Salesman Problem (1-PDTSP)

Samuel Santos Lucas Castilla

Facultad de Ciencias · Sección de Matemáticas

Universidad de La Laguna

alu0100773033@ull.edu.es

Abstract

This memory tackles a variant of the Traveling Saleman Problem, the problem called the Pick-up and Delivery Traveling Salesman Problem, in which starting from a deposit, a number of locations must be visited so that once the tour has been completed, it is returned to the deposit, so that each location must receive or provide a certain amount of product that is transported by a single vehicle, which has a maximum load capacity and in such a way as to minimize the cost of the tour. In this memory, is identified, studied and understood the mathematical methods and models, belonging to the field of combinatorial optimization, through which our problem is effectively resolved.

1. Introduction

Various business sectors, which have a close relationship with logistics processes or localization problems, among others, have a great interest in the use of combinatorial optimization as the main tool for solving their problems since it helps to obtain better results more efficiently. This work is therefore related to the Pick-up and Delivery Traveling Salesman Problem (1-PDTSP). The main theoretical foundations are obtained from book of Salazar González [1]. While the specific concepts of the 1-PDTSP are obtained from the work of Hernández Pérez [2]. In addition, another of the aims of this work has been to ensure that the results obtained support the usefulness of combinatorial optimization to efficiently solve problems within the business world.

2. Theoretical foundations

In the first chapter we deal with the main theoretical concepts for the development of this project. The main purpose of Mathematical Programming is to minimize or maximize a certain objective function in order to satisfy a series of constraints given by the problem to be solved. Then we find a field of research called Routing Problems that are responsible for optimizing a set of transport routes that must be traveled by one or more vehicles from the same warehouse. We briefly explain two types of route problems and their variants.

Traveling Salesman Problem

This problem consists of visiting a number of cities only once by passing through each one of them and starting from any one of them, to which one must return once the route has been completed, and in such a way as to minimise the total distance of the route.

Vehicle Routing Problems

These problems are a generalization of the Travelling Salesman Problem, but where there are also existing and known customer demands that need to be met.

Finally, an important resolution method called "branch-and-cut" is explained in detail. This method is used for the elaboration and implementation of one of the models of the main problem of the project. Finally, the computer tools used are shown.

3. Pick-up and Delivery Traveling Salesman Problem

In the second chapter we focus on the main problem of the project, the Pick-up and Delivery Traveling Salesman Problem. Its main features are as follows: it has a set of locations where an exact and known number of units of goods are collected or delivered by a vehicle with a maximum load capacity, the costs of transporting from one location to another are also known. And each location should be visited only once so as to minimize the total cost. Below, we will show five variants of the 1-PDTSP explaining each of them in detail.

Model with subcycle restrictions

Model with load and flow variables

This second variant is characterized by the use of decision variables, g_{ij}, which defines the load of a dummy flow, of which only one unit must be left on each client, thus preventing the formation of subcycles.

Model with multiflow variable

This variant is characterized because the way to avoid subcycles is to use a multiflow variables defined as g_{ij}^{k} to restrict if you want to reach the destination k by going through the arc (i, j).

Model with potential variable

This variant is characterized by the use of decision variables, u_i, defined as the position of the client i and that makes us avoid the formation of subcycles in the final solution.

Models with capacity restrictions

In this last variant we prevent the formation of subcycles with the following restrictions:

$$\sum_{a \in A(S)} x_a \leq |S| - r(S) \;\; \forall S \subseteq V \;,\; S \neq \emptyset$$

$$\text{with } \; r(S) = \max\{1, \left\lceil \tfrac{|\sum_{i \in S} d_i|}{Q} \right\rceil\}$$

Two different ways, which we have defined as different models in their implementation. In the first one, which we have called models **1-PDTSP capacity**, the model is solved without these restrictions and once the problem is solved, they are inserted if they are violated and then solved again. However, in the second version used **the Branch-and-Cut algorithm**, where the restrictions are inserted first without the need to have the final solution since it is enough to have a fractionated solution.

4. Implementation in Xpress-Mosel and computational results

In the third chapter some important aspects of the implementation of the models are explained, which were studied and analyzed from the Xpress-Mosel manual [3]. The computational times obtained for different models, sizes and capacities, explaining their behavior .

5. Conclusions

Finally, the fourth chapter has been devoted to the presentation of the conclusions obtained during the development of this project.

References

[1] Juan José Salazar González, *Programación Matemática*, Ed. Diaz de Santos, 2001.

[2] Hipólito Hernández Pérez *Procedimientos exactos y heurísticos para resolver problemas de rutas con recogida y entrega de mercancía*, Serie Tesis Doctorales, 2004/05.

[3] FICO: XPress-Mosel (User Guide, Release 4.8), October 2017