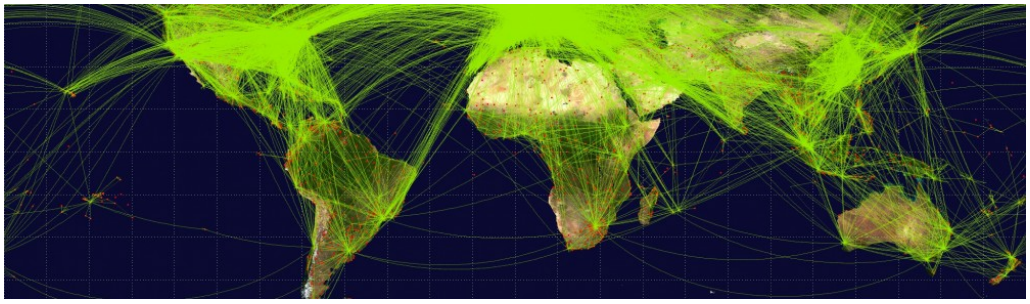
Universidad de La Laguna

Problema del viajante de comercio (TSP) Métodos exactos de resolución

Traveling Salesman Problem (TSP)
Exact solving methods



María Victoria García Travieso

Trabajo de Fin de Grado

Matemáticas, Estadística e Investigación Operativa

Facultad de Matemáticas

Universidad de La Laguna

La Laguna, 16 de julio de 2014

Dr. D. **Juan José Salazar González**, con N.I.F. 43.356.435-D, catedrático de Universidad y Dr. D. **Hipólito Hernández Pérez**, con N.I.F. 45.452.715-T profesor de Universidad adscritos al Departamento de Matemáticas, Estadística e Investigación Operativa de la Universidad de La Laguna

C E R T I F I C A

Que la presente memoria titulada:

"*Problema del viajante de comercio (TSP).*"

ha sido realizada bajo su dirección por D. **María Victoria García Travieso**, con N.I.F. 44.730.704-M.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 16 de julio de 2014

Agradecimientos

A Juan José Salazar y a Hipólito Hernández por su interés y ayuda durante el desarrollo del proyecto.
A Juanfra por esas horas de trabajo juntos para que esto saliera adelante.
A mi familia y a Ana.

Resumen

En muchos sectores de la sociedad se hace necesario, por no decir imprescindible, la optimización de recursos y beneficios, de ahí deriva la importancia que tiene comparar varios modelos matemáticos para resolver determinados problemas de optimización.
En particular, el clásico problema conocido como viajante de comercio se puede enunciar como: "Si un viajante parte de una ciudad y las distancias a otras ciudades son conocidas, ¿cuál es la ruta óptima que debe elegir para visitar todas las ciudades y volver a la ciudad de partida?" Este es un problema de programación entera o mixta. Algunos de los métodos exactos de resolución son ramificación y acotación, hiperplanos de corte y ramificación y corte, siendo este último combinación de los anteriores y de gran interés por reducir considerablemente la cantidad de restricciones objeto de estudio, introduciendo las restricciones de forma dinámica.
Además, estos métodos de resolución se pueden aplicar a diferentes formulaciones del problema como son la formulación con restricciones de subciclo, variables de potencial, variables de flujo y variables de multiflujo. Se hace interesante la comparación mediante los tiempos de ejecución de estos con el de ramificación y corte, utilizando un lenguaje de modelización que se amolde a nuestras necesidades, como es el caso de Mosel, donde se permite la adición de restricciones de forma dinámica mediante el comando setcallback. En esta labor se centra el proyecto titulado Problema del viajante de comercio (TSP), donde además de presentarse los diferentes modelos para el problema del viajante de comercio, se comparan los tiempos de ejecución de los mismos para diferentes tamaños de ciudades. La principal conclusión es que, a pesar, de ser el método de ramificación y corte más rápido, de forma teórica, en este caso el modelo con variables de subciclo da un mejor tiempo computacional. Por lo que se plantea, para proyectos futuros, la posibilidad modificar el planteamiento del subproblema de flujo máximo-corte mínimo contenido dentro del modelo Mosel de ramificación y corte.

Palabras clave: Problema del viajante de comercio, optimización combinatoria, ramificación y corte.

Abstract

In many sectors of society to optimize resources and benefits is necessary, or even essential. That is the reason of the importance of comparing several mathematical models to solve certain problems of combinatorial programming.
In particular, the classic problem known as Traveling Salesman Problem (TSP) can be sketched out as: Ïf a salesman of a city and the distances to other cities are known, what is the best route you should choose to visit all cities and return to the departure city? ". This is a problem of integer or mixed programming. The possible methods of resolution for this kind of problems are branch and bound, cutting planes and branch and cut, the last one is combination of the previous and it has a big interest for significantly reducing the amount restrictions under study, introducing restrictions dynamically.
In addition, there are other methods of resolution using subtour constraints, potential variables, flow variables and multiflow variables, so it is interesting to compare these with the branch and cut under a modeling language able to satisfy our needs, such as Mosel, that is add constraints dynamically by the command: setcallback. This project entitled *Combinatorial Optimization*, where are represented the different models for the Traveling Salesman Problem, compared the execution times of the same models for different sizes of cities.
The main conclusion is that, in spite of being the branch and cut method faster, theoretically, in this case the model of subtour variables gives better computational time. So they show the possibility of focusing certain aspects of branch and cut formulation from another perspective, such as the approach to the problem of *cut minimum* content within the model.

Keywords: *Traveling salesman, combinatorial optimization, branch and cut.*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En el presente proyecto se tratará el problema de programación entera del viajante de comercio (TSP, "Traveling Salesman Problem"). Esto se hará con el fin de comparar diferentes métodos para la resolución de problemas de programación combiantoria, atendiendo a sus tiempos computacionales.

El documento se estructura en seis capítulos diferentes. En el primero de ellos introducimos la motivación que ha llevado al estudio de este tema en particular, los objetivos que se plantean y por último las herramientas utilizadas. En el capítulo dos se muestran una serie de conceptos teóricos necesarios para el desarrollo del proyecto, tratando temas como la programación matemática, su clasificación y los posibles métodos de resolución. En el tercer capítulo nos adentramos en una explicación más extensa del TSP, así como de su formulación matemática tanto para el caso simétrico como asimétrico. En el siguiente capítulo se muestran los comandos necesarios para la implementación del modelo en el lenguaje en el que hemos decidido trabajar. En el capítulo predecesor se dan los resultados computacionales obtenidos. Por último, se dan algunas conclusiones aportando algunas ideas para posibles proyectos futuros.

Tal y como ya se comentó, en el presente capítulo se pretenden mostrar los objetivos principales que se desean abordar con la elaboración de este proyecto de programación matemática, incluyendo así las motivaciones que desembocaron en la elección del mismo. Para ello se introduce cierta terminología que será explicada más adelante. Una vez explicado esto se describe y comparativa de los diferentes optimizadores que permiten abordar los problemas con los que trabajaremos. Así como los motivos principales por los que finalmente se decidió la utilización de Xpress-Mosel.

1.1. Motivación y objetivos

Los problemas de optimización combinatoria aparecen por la necesidad que muestran ciertos sectores empresariales, como los relacionados con procesos de fabricación, logística, problemas de localización, planificación de tareas, etc. Su fin principal es la optimización de uno o varios objetivos, bajo gran números de posibles soluciones. Este efecto hace que sea necesaria la implementación de algoritmos de resolución, capaces de dar soluciones en tiempos computacionales razonables.

Nosotros trabajaremos con el problema del TSP, considerado dentro de los problemas de complejidad alta, es decir, problema del que no se ha encontrado un método de resolución capaz de resolver el problema en un tiempo polinomial. La combinación entre esto y la posibilidad de mejorar los métodos de resolución existentes para reducir los tiempos computacionales han sido las principales motivaciones para el desarrollo del presente proyecto. Además del conocimiento del concepto de optimización gracias a asignaturas como la impartida por la Universidad de La Laguna bajo el nombre de "Programación combinatoria". El presente proyecto tiene como objetivo principal comparar varios modelos matemáticos para resolver un problema clásico de programación combinatoria como es el problema del viajante de comercio (TSP).

Tal y como se muestra en los próximos capítulos existen diversos métodos para la resolución de los diferentes problemas de programación matemática, por lo que cabe la curiosidad de averiguar cuál de ellos, bajo el uso de un lenguaje y un optimizador determinado, sigue un mejor comportamiento frente al resto, es decir, cuál de ellos obtiene una solución óptima en un menor tiempo computacional.

En nuestro caso, por estar en la búsqueda de soluciones enteras y debido a la precisión que los métodos exactos muestran haremos uso de ellos. En especial del Método de Ramificación y corte, por tratarse de una herramienta más reciente, potente y combinación de otros dos métodos que forman parte de este grupo: Ramificación y acotación e Hiperplanos de corte. Este algoritmo será implementado en un entorno llamado Mosel, del cual se darán ciertas características en secciones posteriores.

Un posible método de comparación entre diferentes modelos es ver cuáles son sus tiempos de ejecución para casos concretos. El estudio de estos tiempos computacionales para el método de ramificación y corte, tal y como nosotros los planteamos (mediante la adición de restricciones de forma dinámica), toma especial interés por no ser tan común su comparativa con el resto de métodos de resolución.

1.2. Herramientas

A continuación se muestran las diferentes posibilidades que se barajaron a la hora de determinar el compilador y el software que se emplearían para el desarrollo de este proyecto, así como una descripción de los mismo y las limitaciones que algunos mostraban. Para finalmente centrarnos en aquel que hemos decidido emplear, es decir, en Mosel-Xpress.

En un comienzo se consideró la utilización de un lenguaje algebráico de modelización

llamado "A Mathematical Programming Language"(AMPL). Este es un lenguaje para describir y resolver problemas de programación matemática con alta complejidad. Este lenguaje de modelado es una interfaz que permite llamar a otros optimizadores. Estos optimizadores son:

- CPLEX capaz de resolver algoritmos de los siguientes tipos: lineal mediante el método de resolución del símplex o interior, redes, cuadráticos y entero lineal.
- XLSOL enfocado a la resolución de los siguientes tipos de procedimientos: lineal (símplex), cuadrático y entero lineal.
- MINOS optimizador empleado para aquellos algoritmos clasificados dentro de los lineales (símplex) o no lineales.
- GRG2 diseñado para los algortimos no lineales tanto enteros como no.

En nuestro caso, nos interesamos en el CPLEX, por ajustarse más a nuestras necesidades. Sin embargo, en el proceso de estudio de los diferentes programas nos encaramos con las limitaciones que el propio lenguaje AMPL presentaba para la versión estudiantes. Estas limitaciones estaban relacionadas con el número de restricciones y variables con las que nos permitía trabajar, estando ambas acotadas por una cantidad de 500.

Este hecho forzó el estudio de otros lenguajes como el GUSEK (GLPK Under SCITE Extended Kit) basado en el uso de paquetes de modelización como el GLPK (GNU Linear Programming Kit) y pensado para la programación a gran escala de problemas lineales (LP), enteros mixtos (MIP) y de otros problemas relacionados. Este recurso consta de un conjunto de rutinas escritas en ANSI C (estandarización del lenguaje C con el fin de garantizar su portabilidad entre distintos computadores). Para facilitar su manejo están organizadas en una biblioteca, la cuál puede ser llamada desde el programa.

Este lenguaje presentaba una ventaja añadida relacionada con el previo conocimiento del lenguaje AMPL, ya que GLPK soporta el lenguaje de modelado de GNU MathProg, el cual se corresponde con un subconjunto del lenguaje AMPL.

También se baraja la posibilidad del uso de CMPL: lenguaje programación matemática y optimización de problemas lineales. Este, aunque por defecto, ejecuta CBC es capaz de ejecutar, entre otros, GLPK, tal y como lo hacía GUSEK, pero también existe la posibilidad de utilizar como "solver"el COIN-OR (COmputacional INfrastructure for Operatios Research). Este último es un Open Source, donde los usuarios tienen la capacidad de leer, redestribuir y modificar el código que envuelve al software. COIN-OR ha sido utilizado más comunmente para resolver problemas contenidos en alguno de los siguientes grupos:programación linal, programación no lineal, programación lineal entera mixta,programación no linel entera mixta, programación estocástica lineal, problemas de grafos y combinación de problemas. Los cuales serán explicados en el capítulo 2.

Cuando nos adentramos en el estudio de los mismos nos surgió una nueva necesidad relacionada con el principal objetivo de este proyecto ya que para implementar el algoritmo de ramificación y corte se hace necesaria la posibilidad de añadir las restricciones del problema de forma dinámica. De esta forma nos interesamos en investigar si los recursos con los que estábamos barajando trabajar nos permitían dicha adición. Por la falta de

información acerca de esta cuestión se decidió el uso de Mosel un potente compilador y lenguaje de programación para la rápida optimización del modelo, utilizando el optimizador Xpress que resuelve problemas lineales y enteros mixtos. Este presentaría ciertos problemas, que se explican más adelante, relacionados con el hecho que se corresponde con un lenguaje relativamente nuevo del que, en ciertos aspectos, se carece de información asequible.

1.2.1. Xpress-Mosel

Mosel es un nuevo entorno para el modelado y resolución de problemas, los cuales pueden ser proporcionados tanto en forma de librería como en un programa independiente. Tal y como se comentó en la sección anterior, Mosel es un lenguaje potente capaz de combinar de forma simultánea un modelado y un lenguaje de programación. Es decir, a diferencia de otros entornos como el AMPL donde los problemas se describen en un lenguaje de modelado (declaración de variables, adición de restricciones...), pero las operaciones algorítmicas se presentan en un lenguaje de programación (llamada a un comando de optimización,...), en Mosel no existe separación entre ambas declaraciones. Esta realidad permite programar un algoritmo complejo mediante el entrelazado del modelado y la solución de declaraciones.

Por otra parte, es sabido que cada clase de problema viene asociado a sus propias categorías de variables y restricciones, por lo que la existencia de un único tipo de optimizador para la resolución de todas estas posibles combinaciones se hace ineficiente. Mosel previendo esta situación no integra de forma predeterminada ningún solucionador, pero sí ofrece una interfaz dinámica para "solvers"externos proporcionados en forma de módulos. Cada uno de estos módulos viene dado con su propio conjunto de procedimientos y funciones que extienden de forma directa el vocabulario y capacidades del propio lenguaje. Este vínculo aporta un eficiente enlace entre el Mosel y el "solver"o "solvers"que se emplean en cada caso. A pesar de las grandes ventajas que esto presenta, la más destacada se podría considerar que se corresponde con el hecho que no existe la necesidad de modificar el programa para proporcionar acceso a una nueva tecnología de solución.

En nuestro caso recurriremos al uso del optimizador Xpress, el cual dentro del programa se llama a través del comando *"mmxprs"*. El optimizador Xpress de la compañía FICO es un entorno de programación matemática confeccionado para aportar capacidades de resolución de alto rendimiento. Los problemas pueden ser cargados directamente al optimizador, sin embargo, se recomienda para el usuario la construcción de los problemas usando FICO Xpress Mosel y luego resolver el problema con la interfaz proporcionada por este paquete al optimizador.

Una vez tomada la decisión de hacer uso del "solver"FICO Xpress se hace conveniente realizar una comparativa entre los diferentes optimizadores con los que hemos barajado la posibilidad de trabajar. Para ello utilizaremos una representación gráfica haciendo más visual lo que se quiere expresar en la figura 1.1. Este estudio ha sido realizado por el equipo de SCIP Optimization Suite con el fin de mostrar que su optimizador comercial es el más rápido para la programación entera mixta (MIP) y programación no lineal entera mixta (MINLP), obteniendo los datos de la web de Mittelman (2014), referencia [7]. Pero si nos centramos en el que a nosotros nos concierne podemos observar que no se encuentra en

una mala posición, ya que se corresponde al tercero dentro de los comerciales con los que comparamos. Aunque la versión con la que nosotros trabajaremos será una que ofrece el Xpress sin límite de restricciones ni variables, para un uso no comercial.

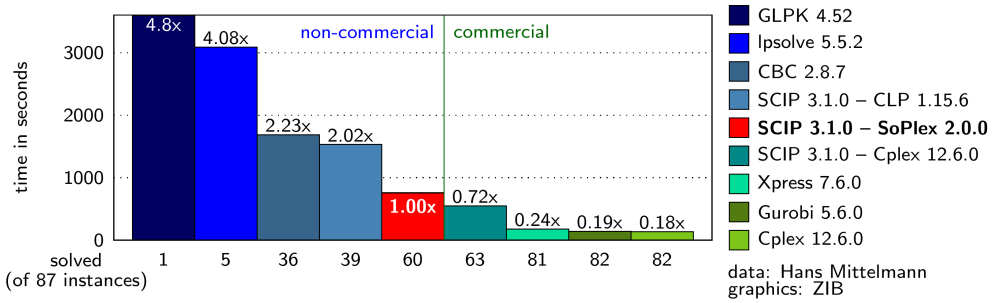


Figura 1.1: Comparativa entre los distintos "solvers"

Capítulo 2

Fundamentos teóricos

En el segundo capítulo del presente documento se dan ciertas nociones básicas del concepto de programación matemática, así como su clasificación atendiendo a las principales características que definen a cada uno de los modelos.

Tras ello nos centramos en la Optimización combinatoria, así como en los diferentes métodos de resolución, enfatisando en los métodos exactos: ramificación y acotación, hiperplanos de corte y ramificación y corte.

2.1. Programación Matemática

La programación matemática forma parte del área de conocimiento de la Investigación Operativa. Se emplea en muchos sectores empresariales durante el proceso de toma de decisiones en las que es necesario satisfacer cierto objetivo bajo una serie de limitaciones dadas o propias de la naturaleza del problema.

La programación matemática se apoya en modelos matemáticos, donde el objetivo a satisfacer se denomina función objetivo, la cual debe ser maximizada o minimizada dependiendo de nuestro propósito principal. Las limitaciones se corresponden con restricciones formuladas como ecuaciones o inecuaciones. En lenguaje matemático, de forma general, este problema puede ser representado como:

$$\text{mín}\{f(x) : x \in S\}$$

donde $S \subseteq \mathbb{R}^n$ y $f : S \to \mathbb{R}$, entendiéndose S como el conjunto donde la función f con la que trabajamos alcanza su valor mínimo. Pasándose a llamar así, a f función objetivo, a cada elemento del conjunto S solución factible y al propio conjunto se le considera región factible.

En el caso de estar interesados en maximizar una función f no tenemos más que tener en cuenta la equivalencia que se muestra, a continuación para, posteriormente, resolver el problema anterior:

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

Debido al carácter general que posee este planteamiento se hace conveniente establecer una clasificación más específica de las diferentes partes que forman la Programación Matemática dependiendo de las características de la función objetivo, variables y ecuaciones/inecuaciones del modelo:

- Atendiendo a las características de las variables:
  - **Programación continua:** todas las variables pueden tomar valores reales.
  - **Programación entera:** se da cuando las variables de estudio solo toman valores enteros. Una parte escencial de este grupo es la llamada ***Optimización Combinatoria*** donde hay un número finito, aunque elevado de posibles decisiones. Este será el grupo principal en el que centraremos nuestra atención durante el presente documento.
  - **Programación mixta:** en este tipo de problemas se da el caso del uso de variables continuas y enteras, es decir, es una combinación de las anteriores.
- Considerando la linealidad de la función objetivo y de las ecuaciones/inecuaciones. En caso de estas ser lineales en las variables se dirá que se trata de un problema de **Programación lineal**, en caso contrario se denominará de **Programación no lineal**.
- Teniendo en cuenta las propiedades de los parámetros asociados al problema:
  - **Programación determinística:** los valores están fijos.
  - **Programación paramétrica:** los parámetros varían de forma sistemática, con la intención de buscar solución para cada uno de estos parámetros.
  - **Programación estocástica:** los parámetros que forman el modelo son variables aleatorias.
- **Programación multiobjetivo:** la función a optimizar es vectorial.

De forma general, los algoritmos de optimización combinatoria resuelven modelos de problemas que se consideran de dificultad alta. Estos algoritmos lo que hacen es reducir el tamaño efectivo del espacio de soluciones y explorar el espacio de búsqueda de manera eficiente.

La mayoría de estos procesos de resolución no garantizan la optimalidad, ni siquiera en el contexto del modelo, ya que se puede dar el caso que no se trate del problema real, pero que su aproximación al óptimo es con una alta probabilidad suficiente. Para entender mejor de lo que se está hablando, a continuación se muestra la clasificación a grandes rasgos de los diferentes métodos de resolución de optimización combinatoria.

2.2. Métodos de resolución para programación entera o mixta

Los métodos de resolución de optimización combinatoria se pueden clasificar de diversas maneras, pero se podría decir que una de las clasificaciones más generales se corresponde

con la siguiente:

- **Método Exacto**: proporciona una solución óptima del problema. Algunos de estos modelos son los que se muestran seguidamente, los cuales serán explicados en el siguiente subapartado.
  - Ramificación y acotación.
  - Hiperplanos de corte.
  - Ramificación y corte.
- **Método heurístico**: algoritmo que con poco esfuerzo computacional proporcionan una solución aproximada, pero no necesariamente óptima del problema. Una clasificación de estos podría ser:
  - Algoritmos constructivos: heurísticos que tratan de construir una solución factible sin precisar de ninguna previa.
  - Heurísticos voraces: hace uso del mejor elemento, teniendo en cuenta el objetivo a optimizar, pero sin tener en cuenta consecuencias futuras, ya que tras tomar una decisión esta no vuelve a ser considerada en futuras iteraciones.
  - Heurísticos de mejora: la idea principal es comenzar con una solución inicial a la que se le quitarán k aristas o arcos para luego añadirle esta misma cantidad de forma que se mejore la solución.
  - Heurístico de multiarranque: trabaja con probabilidades y procesos aleatorios que nos dan mejores soluciones.

Tras este análisis general de los diferentes aspectos que forman la programación matemática y están relacionados con el objetivo del presente proyecto, se hace evidente la necesidad de profundizar más en aquellos métodos de resolución que más nos competen: los métodos exactos. De esta labor nos encargaremos en los siguientes apartados.

En los siguientes puntos se aportan unos conocimientos más extendidos de algunos métodos exactos de resolución. Para ello comenzamos describiendo los algoritmos de Ramificación y Acotación y de Hiperplanos de Corte, para posteriormente definir el método de Ramificación y Corte, explicado en último lugar porque se trata de una combinación de los anteriores.

2.2.1. Ramificación y acotación (Branch and Bound)

Este método comienza con la búsqueda de la solución del problema relajado, es decir, se resuelve el problema sin tener en cuenta el conjunto de restricciones que fuerzan la integrabilidad de las variables, para luego ir añadiendo aquellas limitaciones que nos sean necesarias hasta obtener la solución entera.

Una vez sabido esto podemos decir que su nombre deriva de la representación que se suele dar al conjunto de soluciones que se va obteniendo, ya que éstas usualmente se disponen en forma de árbol, donde cada nodo representa un problema de programación

lineal, poseyendo cada hijo una restricción más que su predecesor o padre. Esta restricción lo que pretende es forzar a que una de las variables obtenidas en la resolución del problema padre sea menor o igual que la parte entera de la solución óptima obtenida para dicha variable, o de forma análoga mayor o igual que la parte entera más uno, formándose así dos nuevos problemas objetos de estudio.

Debido al método seguido en este caso este tipo de algoritmos es considerado de tipo enumerativo, por ir dividiendo y estudiando la región factible hasta la obtención de la solución óptima buscada. El orden seguido para este análisis es el que recibe el nombre de regla FIFO ("First in first out"), es decir, el primero que entre a la lista de problemas a resolver es el primero que se resuelve. Esta lista se irá actualizando a medida que se va avanzando en el árbol de búsqueda.

El procedimiento se dará por finalizado en el momento que ya no queden más elementos objeto de estudio en la lista de problemas a resolver. Esto se puede deber a que a lo largo del árbol decisional se irán actualizando las cotas inferiores y superiores del valor óptimo de las soluciones enteras obtenidas hasta el momento por otras que sean consideradas mejores. De tal forma la solución óptima se dará cuando coindida la cota superior con la inferior. Utilizando el mismo criterio de cotas, podremos parar la ramificación en un nodo determinado si se detecta la no factibilidad o el valor objetivo obtenido es mayor que la cota superior obtenida hasta el momento.

Para comprenderlo mejor, sin dar mucho detalle, la figura 2.1 presenta el diagrama de árbol obtenido con el uso del método de ramificación y acotación:

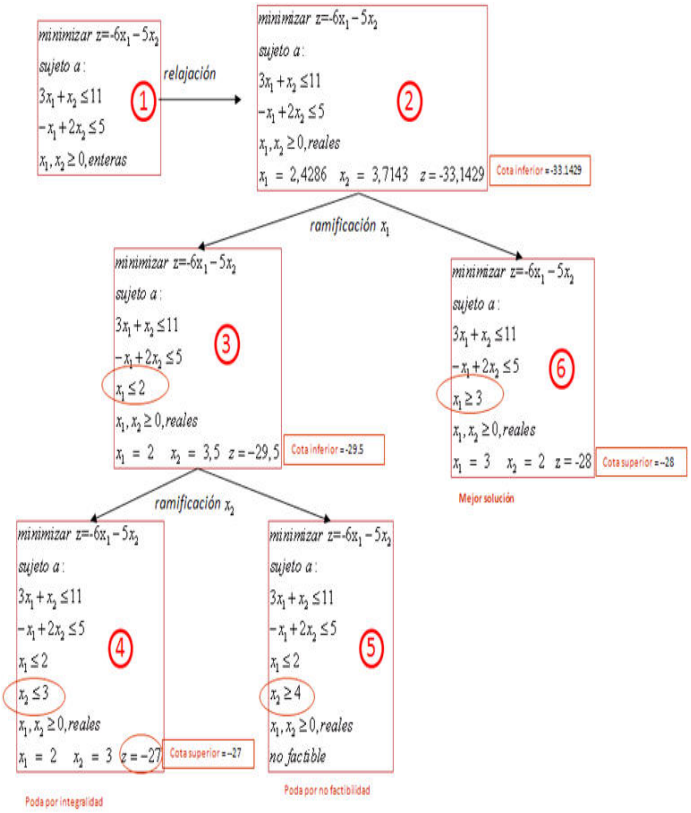


Figura 2.1: Árbol de decisión ramificación y acotación

2.2.2. Hiperplanos de corte (Cutting planes)

Tal y como sucedía con el método anterior este procedimiento se resuelve partiendo inicialmente del problema relajado (i.e. el problema inicial sin las restricciones de integrabilidad). En este caso particular, la técnica seguida es el aislamiento de las soluciones continuas obtenidas de la región óptima del problema, generando un mayor acercamiento a la solución óptima buscada. En otras palabras dado un punto extremo fraccionario x^* de la región factible, se determina el corte o desigualdad $d^T x \leq d_0$ válida para todos los puntos enteros de la región factible y violada por x^*, para luego añadirla al problema. Convirtiéndose así en un método no enumerativo al que también se le conoce como problema de separación.

Este procedimiento se dará por finalizado en el momento en que se detecte en cualquier subproblema del problema original relajado la no acotación, la no factibilidad o la solución obtenida verifique las condiciones de integrabilidad exigida por el problema.

Dentro de este tipo de método uno de los más conocidos es el denominado cortes de Gomory. Este se basa en la desigualdad aportada por Chvátal que si consideramos el problema mín$\{c^T x : Ax = b, x \geq 0yx \in \mathbb{Z}^n\}$ y $P := \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$ poliedro, con A y b compuestos por números enteros, la podemos definir como:

$$\sum_{j=1}^{n} \lfloor u^T a_j \rfloor x_j \leq \lfloor u^T b \rfloor$$

válida para todos los puntos enteros de P, pero no tiene por que serlo para el resto de puntos.

2.2.3. Ramificación y corte (Branch and cut)

Este nuevo método aparece por la necesidad de acelerar el proceso de resolución de problemas de programación lineal entera. Ya que el método de hiperplanos de corte a medida que vamos aumentando el número de interaciones el aporte del nuevo corte comienza a ser insignificante, y el método de ramificación y acotación corre con la desventaja de la necesidad de estudiar un número elevado de nodos del árbol decisional generado, debido a la usual diferencia que se da entre la solución óptima continua y la entera. En este caso, a pesar de emplearse un tiempo ligeramente mayor para el estudio de cada nodo, el número de nodos objeto de estudio es considerablemente menor, debido a la reducción del número de restricciones añadidas.

El procedimiento empleado, de forma general, consiste en ir añadiendo cortes a medida que avanzamos en el árbol decisional. Estos cortes son específicos y no se asegura su convergencia a una solución óptima. Los pasos seguidos son: resolución del problema relajado, resolución del problema de separación añadiendo la restricción pertinente, repitiendo este último paso hasta que este método no añada ninguna restricción más por ser cumplidas todas las establecidas hasta el momento. Para ver que estas son cumplidas se emplea un procedimiento externo polinomial. En este punto nos podemos enfrentar a dos situaciones: se obtenga la solución óptima entera o, por el contrario, se incumpla la condición de integrabilidad. En cuyo caso se procede a la implementación del método de ramificación y acotación. Una vez hecho esto retomaremos el proceso de separación. Se realiza este algoritmo hasta obtener la solución entera buscada, o se detecte la no acotación o no optimalidad siguiendo el criterio empleado en el método que estemos empleando en el momento de la detección.

Capítulo 3

Problema del viajante de comercio (TSP)

3.1. Introducción

Problema del Viajante de Comercio, del inglés Traveling Salesman Problem o también conocido como TSP, de origen desconocido, pero debiendo su nombre a la comunidad científica (1931-1932) de la Universidad de Pincenton, donde se toma el problema desde un punto de vista matemático.

El Problema del Viajante de Comercio es uno de los problemas más complejos que se conoce de la programación matemática actual por su complejidad computacional, estando clasificado dentro de aquellos problemas considerados *NP-Duro* porque, a día de hoy se ha demostrado que tal método polinomial no existe (salvo que se llegue a demostrar que P=NP).

La importancia del TSP no sólo radica en la cantidad de aplicaciones que tiene sino que, también, la investigación realizada sobre el TSP es fácilmente aplicable a otros problemas de rutas que se suelen generalizar a este último. Un ejemplo de ello lo encontramos en problemas de rutas con recogida y entrega de mercancías (ver Hernández (2004/05) [8]). Este hecho ha favorecido a que se trate de uno de los más estudiados en el campo de la Investigación Operativa, con unos records historicos computacionales tales como los que se muestran en la tabla 3.1.

Año	**Autores**	**Ciudades**
1954	Dantzig, Fulkerson, and Johnson	49
1971	Held and Karp	64
1975	Camerini, Fratta, and Maffioli	67
1977	Grötschel	120
1980	Crowder and Padberg	318
1987	Padberg and Rinaldi	532
1987	Grötschel and Holland	666
1987	Padberg and Rinaldi	2,392
1994	Applegate, Bixby, Chvátal, and Cook	7.397
1998	Applegate, Bixby, Chvátal, and Cook	13.509
2001	Applegate, Bixby, Chvátal, and Cook	15.112
2004	Applegate, Bixby, Chvátal, Cook, and Helsgaun	24.978
2005	Cook, Espinoza and Goycoolea	33.810
2005/06	Applegate, Bixby, Chvátal, and Cook	85.900

Tabla 3.1: Recórds históricos para el problema del TSP

De los datos recopilados en la tabla 3.1 cabe destacar el hallazgo hecho en 2006, con 85900 ciudades. Este venía motivado porque a finales de 1980 gracias a una aplicación VLSI (Very Large Scale Integration) creada en los laboratorios de Bell se plantea resolver el problema el TSP para 85900 nodos. En este caso los investigadores estaban interesados en minimizar el tiempo total que empleaba un láser en la elaboración de chips. Considerando la ciudades como las ubicaciones de las interconexiones a cortar, y el costo del viaje entre las ciudades al momento de pasar de una interconexión a otra. Pero no fue hasta 2006 cuando se obtuvo el orden en el que el láser debe cortar las interconexiones en las que estamos interesados, minimizando el momento de pasar de una interconexión a otra. Si se desea ampliar la información sobre estos hechos se recomienda la lectura de Applegate, Bixby, Chvátal y Cook (2006) [1] donde se refleja este último resultado, y como curiosidad se recomienda la lectura del primer artículo donde se plantea un modelo matemático para el TSP: Dantzig, Fulkerson, Johnson (1954) [4]

Este problema es reconocido como un problema de rutas, definidos como problemas de optimización generados cuando existen unos clientes que demandan un servicio y se debe localizar la ruta óptima para satisfacerles. Otras aplicaciones para este problema son la recogida de basura, entrega de correo, transporte de mercancías, llegando incluso a terrenos como la producción de circuitos eléctricos o la secuenciación de tareas. La causa principal de este interés es el ahorro que supone la optimización de recursos.

En esta sección nos centramos principalmente en la descripción del problema clásico del Viajante de Comercio, así como de las posibles variantes que han ido apareciendo tras su planteamiento inicial. Tras haber comprendido este concepto se introducen varios modelos, tanto para el caso simétrico como para el asimétrico: modelo con restricciones de subciclo, modelo con variables de flujo, modelo con variables de multiflujo y modelo con variables de potencial.

3.2. Planteamiento del problema

Existen diversos tipos de problemas de rutas, los cuales están diferenciados por las restricciones que se impongan. En nuestro caso, debemos añadir todas aquellas limitaciones que hagan cumplir lo que el propio enunciado del problema requiere. Un posible enunciado general del problema podría ser el dado por Calviño (2011) [6] donde enuncia:

"Si un viajante parte de una ciudad y las distancias a otras ciudades son conocidas, ¿cuál es la ruta óptima que debe elegir para visitar todas las ciudades y volver a la ciudad de partida?"

En otras palabras lo que vamos buscando es que si tenemos un vendedor que debe visitar una cierta cantidad n de ciudades, pasando por cada ciudad una única vez, donde:

- El punto de partida se corresponde con la base o depósito, por lo que esta ciudad debe venir prefijada, y además, corresponderse al destino final del recorrido, siendo así un circuito cerrado.
- El grafo formado por las diferentes ciudades puede ser dirigido o no dirigido, diferenciándose por el hecho de que las conexiones entre ciudades tengan dirección o no, respectivamente.
- Los costes o las distancias entre las diferentes ciudades son conocidos, puediendose representar mediante una matriz cuadrada n x n. Matriz simétrica o asimétrica dependiendo de si el grafo es no dirigido o dirigido, respectivamente.

Siendo el objetivo final encontrar el orden en el que deben ser visitadas las ciudades con el fin de minimizar el coste o distancia de la ruta.

Finalmente, lo que obtendremos será un ciclo simple o un circuito hamiltoniano el cual se define como:

Definición: *Sea un grafo G se dice que un circuito o ciclo hamiltoniano es un ciclo simple que contiene a todos los vértices de G. Equivalentemente es una trayectoria que empieza y termina en el mismo vértice, no tiene aristas repetidas y pasa por cada vértice una única vez.*

De esta forma, para unos ciertos nodos en particular la solución óptima, atendiendo a la optimización del coste, se podría representar como se muestra en la siguiente imagen:

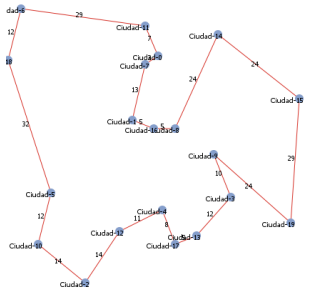


Figura 3.1: Ejemplo de circuito hamiltoniano

A lo largo de la historia del TSP se han generado numerosas variantes del mismo problema, entre las que se podrían destacar:

- Problema del Ciclo Simple (SCP, Simple Cycle Problem): el ciclo solución no tienen por qué estar incluidas todas las ciudades. Esto se debe a que, para este caso, no solo existen costos por visitar las ciudades sino también hay beneficios por cada ciudad.
- Problema del viajante de comercio generalizado (GTSP, Generalized Traveling Salesman Problem): el conjunto de ciudades está dividido en regiones, y debemos visitar una ciudad de cada región minimizando el coste.
- Problema del viajante de comercio con recogida y entrega de mercancías (PDTSP, Pick-up and Delivery Traveling Salesman Problem): cada ciudad provee o recibe una cantidad de producto que es transportada por un único vehículo con capacidad limitada.
- Problema del viajante de comercio con ventanas de tiempo (Traveling Salesman Problem with Time Windows): donde cada ciudad o cliente tiene un tiempo de mínimo de llegada y máximo de salida.
- Problema del viajante de comercio con múltiples viajantes: existe un número determinado de viajantes que deben visitar ciertas ciudades, sin visitar las que ya han sido visitadas por el resto de viajantes.

Una vez tenemos la descripción y los diferentes tipos de TSP procedemos a introducir este mismo problema pero en lenguaje matemático. Tal y como se enunció anteriormente, en el problema general, estamos interesados en que no se generen subciclos, es decir, nuestro fin es visitar todas las ciudades una única vez formando un ciclo que comience y termine en el mismo punto. Para evitar la aparición de estas conexiones aisladas existen varios métodos.

En nuestro caso los diferentes modelos del problema se han formulado para el caso asimétrico, a pesar de trabajar con datos simétricos (donde la distancia entre la ciudad i y la j es igual a la distancia entre j y la i, para cualquier i, j de las n ciudades), ya que este presenta un carácter más general. Sin embargo, con fin didáctico se introducirá tanto el caso simétrico como el asimétrico. Para la formulación del problema es necesario cierto conocimiento de teoría de grafos, de la que si se desea ampliar información se recomienda Salazar (2001) [3].

3.2.1. Formulaciones para el caso Asimétrico

Nos enfrentamos al caso en el que la distancia entre i y j no coincide con la distancia de j a i. Por lo que siguiendo la teoría de grafos definimos grafo dirigido como $G = (N, A)$, donde N es el conjunto de ciudades $N := \{1, ..., n\}$ y A el conjunto de arcos. $\delta^+ := |\{j \in N : (i, j) \in A\}| \equiv$ número de arcos que tienen a i como origen y por último $\delta^- := |\{j \in N : (j, i) \in A\}| \equiv$ número de arcos que tienen a i como destino.

Cada arco a puede ser denotado como el par (i, j), teniendo asociado un coste al cual denotaremos como c_a. Se suele considerar que G es un grafo completo, es decir, que el conjunto de arcos A tiene todas las posibles arcos, ya que si este no fuera el caso le pondríamos un coste elevado $(\sum_{a \in A} c_a)$ obligando a que este arco no sea elegida dentro de la solución óptima, por esta ir en busca del costo mínimo.

Para formular el problema debemos considerar una variable decisional x_a , asociado a cada uno de los arcos $a \in A$, expresada como:

$$x_a := \begin{cases} 1 & \text{si el arco } a \text{ forma parte de la solución óptima} \\ 0 & \text{otro caso} \end{cases}$$

Modelo con restricciones de subciclo

En este caso, además, debemos hacer uso del subconjunto S estrictamente contenido en N, que será en el que forzaremos que no hayan subciclos.

$$\text{mín} \sum_{a \in A} c_a x_a \tag{3.1}$$

Sujeto a:

$$\sum_{a \in \delta^+(i)} x_a = \sum_{a \in \delta^-(i)} x_a = 1 \quad \forall i \in N \tag{3.2}$$

$$\sum_{a \in A(S)} x_a \leq |S| - 1 \quad \forall S \subset N \tag{3.3}$$

$$0 \leq x_a \leq 1 \tag{3.4}$$

$$x_a \in \mathbb{Z} \tag{3.5}$$

Las restricciones (3.2) fuerzan a que, únicamente, entre un arco y salga otro de cada nodo, esto viene motivado por las restricciones (3.5) y (3.4) que fuerzan a que las variables solo puedan tomar valores binarios. Las restricciones (3.3) evitan la aparición de subciclos,

forzando que en cualquier subconjunto de nodos contenido estrictamente en S no pueda haber más arcos que el número de nodos que contiene menos uno, obligando así a que de ese subconjunto salga al menos una arista. Estas últimas serán las que iremos modificando en los diferentes métodos posibles para evitar subciclos.

Modelo con variables de potencial

Para este planteamiendo debemos introducir la variable entera u_j que determina la posición del nodo j en la ruta a determinar, quedando así un planteamiento:

$$\text{mín} \sum_{a \in A} c_a x_a \tag{3.6}$$

Sujeto a:

$$\sum_{a \in \delta^+(i)} x_a = \sum_{a \in \delta^-(i)} x_a = 1 \qquad \forall i \in N \tag{3.7}$$

$$u_j \leq u_i + x_{ij}–(n-1)(1-x_{ij}) + (n-2)x_{ji} \quad \forall i, j \in V \setminus \{1\} \tag{3.8}$$

$$0 \leq x_a \leq 1 \tag{3.9}$$

$$x_a \in \mathbb{Z} \tag{3.10}$$

Como se puede apreciar continuamos manteniendo las mismas restricciones, exceptuando las de subciclos, ya que en este caso, las restricciones 3.8 lo que hacen es forzar a que, partiendo de una ciudad determinada y llegando a esta misma, el orden seguido para visitar a las diferentes sea coherente. Esto se produce porque en caso de existir un subciclo aislado si consideramos la ciudad i de este subciclo y le asignamos el orden 2, por ejemplo, cuando hayamos recorrido todos los nodos que forman el subciclo , pongamos que sean 5, e intentemos volver al 2 le deberíamos asignar el orden 6, sin embargo, esto contradice al conjunto de restricciones que estamos imponiendo.

Modelo con variables de flujo

En este caso haremos uso de una variable de flujo f_{ij}, muy parecida a la de potencial, pero inversa, es decir en este caso no vamos sumándole una unidad cada vez que visitamos una ciudad, sino que por el contrario se la restamos. Formulándose como se muestra a continuación:

$$\text{mín} \sum_{a \in A} c_a x_a \tag{3.11}$$

Sujeto a:

$$\sum_{a \in \delta^+(i)} x_a = \sum_{a \in \delta^-(i)} x_a = 1 \qquad \forall i \in N \tag{3.12}$$

$$\sum_{j \in V, j \neq i} f_{ji} - \sum_{j \in V, j \neq i} f_{ij} = 1 \quad \forall i \in V \setminus \{1\} \tag{3.13}$$

$$f_{1i} = (n-1)x_{1i} \quad \forall i \in V \setminus \{1\} \tag{3.14}$$
$$f_{i1} = 0 \quad \forall i \in V \setminus \{1\} \tag{3.15}$$
$$f_{ij} \leq (n-1)x_{ij} \quad \forall i \in V \setminus \{1\} \tag{3.16}$$
$$0 \leq x_a \leq 1 \tag{3.17}$$
$$x_a \in \mathbb{Z} \tag{3.18}$$

Con las restricciones (3.13), (3.14), (3.15) y (3.16) lo que se pretende es que cada vez que pasemos por un nodo descarguemos una unidad de la variable flujo, obligando a que del nodo de partida se salga con un flujo igual al número de ciudades menos uno, que no se regrese al nodo de partida con flujo y que solo exista flujo entre dos ciudades si existe la conexión entre estas.

Modelo con variables de multi-flujo

Aquí también haremos uso de una variable flujo, pero en este caso no solo atendemos a la conexión entre i y j sino que, además, consideramosun producto diferente para cada nodo k, apareciendo variables como f_{ij}^k, $i, j, k \in N$. Con un modelo como:

$$\text{mín} \sum_{a \in A} c_a x_a \tag{3.19}$$

Sujeto a:

$$\sum_{a \in \delta^+(i)} x_a = \sum_{a \in \delta^-(i)} x_a = 1 \qquad \forall i \in N \tag{3.20}$$
$$\sum_{j \in V, j \neq 1} f_{1j}^k = 1 \qquad \forall k \in V \setminus \{1\} \tag{3.21}$$
$$f_{j1}^k = 0 \qquad \forall k \in V \setminus \{1\} \tag{3.22}$$
$$\sum_{j \in V, j \neq k} f_{jk}^k = 1 \qquad \forall k \in V \setminus \{1\} \tag{3.23}$$
$$f_{kj}^k = 0 \quad \forall j, k \in V \setminus \{1\}, j \neq k \tag{3.24}$$
$$\sum_{j \in V, j \neq i} f_{ji}^k - \sum_{j \in V, j \neq i} f_{ij}^k = 0 \quad \forall i \in V \setminus \{1\}, i \neq k \tag{3.25}$$
$$f_{ij}^k \leq x_{i,j} \qquad \forall i, j, k \in V \setminus \{1\} \tag{3.26}$$
$$0 \leq x_a \leq 1 \tag{3.27}$$
$$x_a \in \mathbb{Z} \tag{3.28}$$

3.2.2. Formulaciones para el caso Simétrico

Como ya se dijo anteriormente este caso se corresponde al caso en el que el coste o la distancia (depende de lo que estemos interesados en optimizar) para conectar i y j sea idéntico que el de conectar j con i, reduciéndose así la cantidad de variables a la mitad con

respecto al planteamiento para el caso asimétrico. De este modo y atendiendo a la teoría de grafos podemos considerar el grafo no orientado $G = (N, E)$, donde N se corresponde con el conjunto de nodos y E con el de aristas. Con un coste c_e asociado a cada arista $e \in E$. Se suele considerar que G es un grafo completo, es decir, que el conjunto de aristas E tiene todas las posibles arista, ya que si este no fuera el caso le pondríamos un coste elevado ($\sum_{e \in E} c_e$) obligando a que esta arista no sea elegida dentro de la solución óptima, por esta ir en busca del costo mínimo.

El modelo matemático para el caso simétrico es:

$$x_e := \begin{cases} 1 & \text{si la arista } e \text{ forma parte de la solución óptima} \\ 0 & \text{otro caso} \end{cases}$$

De este modo, si definimos $\delta(i) := \{[i, j] \in E : j \in N\}$ y consideramos como nodo de partida el 1 podemos enunciar los diferentes modelos para el TSP atendiendo a las restricciones o a las variables que se añaden para la eliminación de subciclos.

Para simplificar lo que haremos será enunciar el problema principal pero sin añadirle las restricciones de subciclos, ya que estas se mantienen exactamente igual que para el caso simétrico. En este caso haremos uso de la variable:

$$\text{mín} \sum_{e \in E} c_e x_e \tag{3.29}$$

Sujeto a:

$$\sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in N \tag{3.30}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad \forall S \subset N \tag{3.31}$$

$$0 \leq x_e \leq 1 \tag{3.32}$$

$$x_e \in \mathbb{Z} \tag{3.33}$$

$$\tag{3.34}$$

NOTA: La implementación de estos métodos en el entorno Mosel serán añadidos en el Anexo A.

Capítulo 4

Implementación del modelo en Mosel. Tiempos de computacionales.

En el presente capítulo se introducen todos aquellos aspectos que se hicieron necesarios para el modelado en Mosel del problema del viajante de comercio, principalmente, para el método de ramificación y corte. Además se enuncia, de forma breve, la estructura seguida para la ejecución del modelo.

Como ya se ha comentado en capítulos anteriores uno de nuestros objetivos era conseguir que Mosel insertara, cada vez que fuera necesario, restricciones de forma dinámica (i.e. que el programa permitiese la incorporación de determinadas restricciones cada vez que se incumplieran ciertos requisitos propios del problema), ya que este es un requisito necesario para la implementación del método de ramificación y corte. Este hecho llevó al estudio de los diferentes recursos que mostraba el entorno para este fin. De entre todas las posibilidades barajadas se decidió la utilización de unos comandos determinados, los cuales se presentan en la siguiente sección.

4.0.3. Restricciones dinámicas. Procedimientos para la implementación

La adición de restricciones de forma dinámica no es un recurso muy empleado, sin embargo, este presenta grandes ventajas, como puede ser la reducción del número de restricciones que deben ser consideradas, ya que solo se considerarán, en cada momento, aquellas que sean imprescindibles. Este hecho lleva una ventaja vinculada ya que, de forma general, debe reducir el tiempo computacional.

Xpress-Optimizer Reference Manual (2009) [10] refleja que para efectuar esta labor es necesario el uso del recurso `setcallback`, el cuál llama de forma recursiva en diferentes puntos del proceso ciertas funciones o procedimientos definidas por el usuario. Esta llamada tiene una expresión tal y como la que se muestra a continuación:

```
Setcallback(\...tipo de llamada recursiva...", \...nombre de la función o
                                procedimiento...")
```

Existen numerosos tipos de llamadas, los cuales vienen asociados a determinadas funciones y procedimientos, cada uno de ellos para una labor diferente. Pero en la que nosotros nos centraremos es en la del tipo `XPRS_CB_CUTMGR`, la cual bajo una función booleana es capaz de controlar los cortes añadidos en el método de ramificación y corte.

Otra de las rutinas sumamente importantes para la implementación del modelo es aquella que añade el corte o los cortes al nodo actual y a sus descendientes. Esta es la ejecutada según:

```
Addcuts(\...identificación del corte...",\...tipo de
          desigualdad...",\...corte...")
```

A lo largo, del proceso de ejecución se hace necesario el uso de otra `setcallback`, pero en este caso para la detección de soluciones enteras y el posterior rechazo, si esta no se amolda al resultado que nosotros buscábamos, es decir, si contiene algún subciclo, o por el contrario, para su aceptación como solución válida. En este caso, estamos hablando de `setcallback(XPRS_CB_PREINTSOL,\...procedure cb(isheur:boolean,cutoff:real...")`. Una vez visto esto se hace interesante comprender cómo se programa este método.

4.0.4. Implementación del método de ramificación y corte

Para la implementación del método de ramificación y corte se sigue el siguiente algoritmo:

1. Entraremos en este proceso siempre y cuando el programa decida que es necesaria la adición de un corte, por tanto la función *addcut* nombrada anteriormente debe colocarse a verdadera.

2. Creamos un grafo con ciertas cargas, las cuales se corresponden con la solución parcial del problema.

3. Buscamos las restricciones violadas a partir de un procedimiento de complejidad polinomial. En el TSP se considera que las restricciones violadas son aquellas que no cumplen que, dado un subconjunto del conjunto de nodos, el número de arcos que sale de este subconjunto es mayor o igual a 1. En nuestro caso, resolvemos el problema de flujo máximo, por demostrarse que este es equivalente al problema del corte mínimo. Esto supondría optimizar dos problemas al mismo tiempo, hecho que Mosel, de forma natural, no es capaz de hacer, por lo que hacemos uso del recurso: `\with mpproblem do...`".

4. Añadir los cortes necesarios detectados con ayuda del problema de flujo máximo, mediante `addcuts` y gestionando su adición recursiva con la función `XPRS_CB_CUTMGR`, añadiendo solo aquellas que nos son útiles.

5. Repetir este proceso hasta que no se encuentre ninguna restricción violada.

A lo largo del proceso se irá añadiendo una cantidad considerable de restricciones, por lo que conviene de forma periódica detectar cuáles no son útiles para dejar de hacer uso

de ellas, al menos temporalmente, ya que puede darse el caso que vuelvan a ser necesarias momento en el que las volveríamos a considerar.

NOTA: A lo largo de esta función se hace necesaria la utilización de un parámetro con un valor próximo a cero que evite errores de tolerancia. Para una mejor comprensión de lo que se pretende expresar se recomienda ver el Anexo A.5.

Capítulo 5

Resultados computacionales

En el presente capítulo, se expondrá la descripción e importancia de los resultados computacionales. Para, finalmente, exponer los datos obtenidos para los diferentes métodos enunciados en los dos capítulos anteriores.

Ya en el primer capítulo del presente proyecto se presentaron las ideas básicas y el interés que tenía la obtención de los resultados computacionales. De este modo a lo largo de la historia se han realizado numerosos estudios donde se comparan diferentes métodos de ejecución para problemas determinados o bien se intenta mejorar un proceso reduciendo este tiempo con respecto a resultados anteriores para poder aumentar de forma considerable el número de ciudades y/o conexiones, en el caso del TSP.

Antes de introducir los datos computacionales se hace conveniente exponer las características principales del sistema en el que se ha ejecutado. En nuestro caso se trata de un ordenador con un sistema operativo Windows7 de 64 bits, un procesador Pentium(R) Dual-Core CPU con velocidad 2.20GHz. Tiene una memoria instalada (RAM) de 4GB, de los cuales 1.5GB aproximadamente son de memoria libre, 512MB de gráficas y 2GB de memoria de sistema.

En nuestro caso hemos decidido utilizar cantidades de ciudades con las que somos capaces de trabajar, por estar tratando con diferentes métodos. Tomando los datos para 25, 50, 75, 100 y 125 ciudades, cambiando cinco veces de semilla para la aleatoriedad de los datos. Las ubicaciones de las ciudades han sido generadas de forma aleatoria, creando las coordenadas a partir de redondear el producto entre un número random y el número de ciudades, de forma que estas coordenadas en el rectángulo [0,n]x[0,n], donde los costos entre cada par de localizaciones corresponde con la distancia euclídea redondeada.

En la tabla 5.1 la primera columna n corresponde al número de ciudades en cada ejemplo, la segunda “seed ”son las semillas utilizadas en la generación de cada ejemplo y las cinco últimas columnas son los tiempos de ejecución de cada uno de los procedimientos utilizados. Los modelos con variables de flujo (“flow”), multiflujo (“multiflow”) y potencial (“potencial”) tienen en común que resuelven el problema completo, es decir, añaden directamente las restricciones de subciclo al problema original. Sin embargo, el modelo con restricciones de subciclo (“subtour”) lo que hace es resolver el problema sin las restricciones para la eliminación de subciclos, para luego ser añadidas según se vayan considerando

necesarias. Este método también es seguido por el modelo de ramificación y corte, pero, además, este trabaja sin las restricciones de integrabilidad, es decir, comienza con la resolución del problema relajado. Estos factores condicionaran los tiempos de ejecución de los diferentes modelos.

A continuación se muestran los datos obtenidos, teniendo en cuenta que al llegar a los 1800s el procedimiento para (i.e. se considera un tiempo límite de 1800s):

n	**Seed**	**Subtour**	**Flow**	**Multiflow**	**Potencial**	**Branch-cut**
25		**1,8**	**6,8**	**6,5**	**6,9**	**0,6**
	A	1,5	1,1	5,1	9,1	0,7
	B	1,6	7,8	7,8	4,3	0,6
	C	2,6	20,6	5,6	4,1	0,4
	D	2,2	3,4	8,2	9,4	1,1
	E	0,9	1,3	5,0	7,8	0,2
50		**18,5**	**752,6**	**369,8**	**813,7**	**15,0**
	A	10,0	120,2	204,4	166,2	1,6
	B	27,7	1800,0	393,4	1800,0	17,2
	C	14,0	32,0	424,1	251,4	10,6
	D	26,7	1800,0	295,1	1800,0	37,5
	E	14,1	11,1	531,9	51,0	8,3
75		**63,9**	**1155,0**	**1800,0**	**1800,0**	**101,5**
	A	59,7	1800,0	1800,0	1800,0	68,5
	B	83,7	1800,0	1800,0	1800,0	45,4
	C	79,1	325,8	1800,0	1800,0	332,1
	D	23,6	49,1	1800,0	1800,0	9,5
	E	73,2	1800,0	1800,0	1800,0	51,9
100		**144,2**	**1800,0**	**1800,0**	**1800,0**	**425,4**
	A	164,0	1800,0	1800,0	1800,0	839,6
	B	146,8	1800,0	1800,0	1800,0	609,9
	C	128,9	1800,0	1800,0	1800,0	298,9
	D	124,0	1800,0	1800,0	1800,0	64,5
	E	157,3	1800,0	1800,0	1800,0	314,1
125		**435,8**	**1800,0**	**1800,0**	**1800,0**	**1722,3**
	A	525,2	1800,0	1800,0	1800,0	1411,6
	B	480,9	1800,0	1800,0	1800,0	1800,0
	C	459,6	1800,0	1800,0	1800,0	1800,0
	D	362,6	1800,0	1800,0	1800,0	1800,0
	E	351,0	1800,0	1800,0	1800,0	1800,0
Total Resultados		**132,8**	**1102,9**	**1155,3**	**1244,1**	**453,0**

Tabla 5.1: Tiempos computacionales

En la anterior tabla se encuentran los diferentes datos obtenidos, así como los tiempos promedios de ejecución. En la misma se puede apreciar cuál es la evolución de los diferentes métodos, viéndose claramente que, de forma general, el procedimiento menos recomendable es del de variables de potencial y el de restricciones de subciclo el que, con diferencia, sigue un mejor comportamiento. También se pone de manifiesto cómo para tamaños pequeños el ramificación y corte es el que mejor comportamiento tiene, sin embargo a medida que aumentamos este tamaño quien pasa a ser más efectivo es el método de subciclo. Esto ha sido un resultado inesperado, ya que teóricamente, el modelo de ramificación y corte debe ser el que siempre aporte un mejor resultado computacional. La situación contraria puede ser causada porque el procedimiento de separación implementado resuelve un problema de programación lineal que tiene que resolverse iterativamente con tamaños grandes (i.e. $n = 100$ y $n = 125$). Sin embargo, el modelo con restricciones de subciclo para estos mismos tamaños el Xpress resuelve los problemas enteros de forma muy rápida, a pesar de que para llegar a la solución óptima necesite resolver muchos de problemas de forma reiterada.

Con los procedimientos de flujo y multiflujo sucede algo parecido, ya que en un principio el multiflujo tiene un menor tiempo de ejecución, pero cuando consideramos 50 ciudades se intercambian los papeles, y al aumentar a tamaños mayores o igual a 75 ambos requieren tiempos de ejecución elevados.

Por último, cabe mencionar que el método de potencial, para los casos estudiados no es recomendable ya que es el que mayor tiempo computacional presenta en todo momento.

Una representación más visual de los datos obtenidos la muestra la siguiente gráfica, donde se puede apreciar cada uno de los aspectos que se han tratado anteriomente.

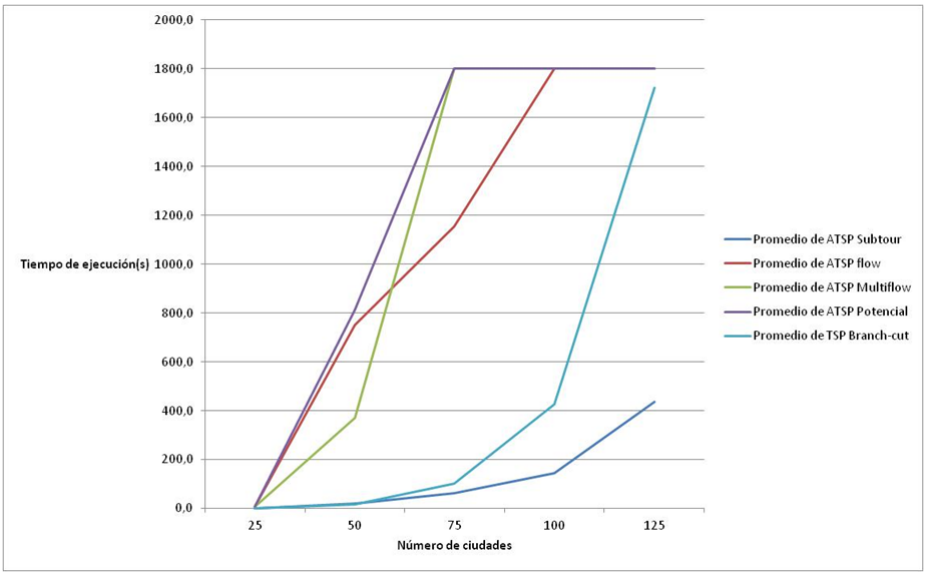


Figura 5.1: Gráfica de tiempos computacionales

Capítulo 6

Conclusiones

Como punto innovador del proyecto *"Problema del viajante de comercio (TSP)"* podemos destacar el uso del comando Mosel: *setcallback*. Este ha sido crucial para la implementación del modelo de ramificación y corte, permitiendo reducir considerablemente el tiempo computacional para la resolución del problema del TSP con respecto a modelos como los de variables de flujo, multiflujo o potencial.

Como contrapartida tenemos el hecho de el modelo de ramificación y corte no es más eficiente que el modelo con restricciones de subciclo para tamaños grandes. Esto motiva a mejorar el modelo presentado, ya que, teóricamente, el método de ramificación y corte debe requerir un menor tiempo computacional, por utilizar un menor número de restricciones. Como posible mejora se recomienda modificar la forma de plantear el subproblema de flujo máximo-corte mínimo, ya que a pesar de ser un proceso lineal y capaz de resolverse en tiempo polinomial, el tener que utilizar un resolutor lineal para este problema puede ralentizar el proceso. Existen algoritmos más eficientes para el problema del flujo máximo-corte mínimo, pero en este trabajo no se han implementado ya que éste no era el objetivo del mismo.

El proceso de investigación seguido para la elaboración del proyecto ha sido en ocasiones lento e incluso frustrante, pero aún así satisfactorio. Una puerta a la investigación y a todo lo que ello conlleva, en particular, dentro de un área en la que numerosos sectores de la sociedad emplean recursos por los grandes beneficios que puede aportar el estudio de la investigación operativa.

Otro hecho que se deja entrever es cómo este proyecto puede ser de ayuda para futuras investigaciones, tal y como lo han sido resultados ya obtenidos para el desarrollo de este. Todos los que trabajamos en el área formamos la escalera que lleva a los resultados buscados.

Apéndice A

Scripts de los métodos de resolución

A.1. Programa Mosel con restricciones de subciclo

```
(!*****************************************************************************************************
TSP: resolución del problema del TSP formulación con restricciones de subciclos
*****************************************************************************************************!)

!Se cargan las librerías y se le da nombre al modelo
!-----------------------------------------------------------------------------------------------------
model "TSP subtour"
uses "mmxprs","mmive","mmsystem"

!Se advierte de la existencia de determinadas funciones y procedimientos que se encuentran más adelante
!-----------------------------------------------------------------------------------------------------
forward procedure break_subtour
forward procedure draw

declarations
NCITIES = 25
seed = 4

x: array(1..NCITIES) of integer
y: array(1..NCITIES) of integer

CITIES = 1..NCITIES ! Cities
DIST: array(CITIES,CITIES) of integer
NEXTC: array(CITIES) of integer
assig: array(CITIES,CITIES) of mpvar
mips,points,roads:integer
now1, now2: datetime
end-declarations

!Se establecen los parámetros de forma aleatoria
!-----------------------------------------------------------------------------------------------------
setrandseed(seed)
```

```
forall(i in CITIES) do
  x(i):=round(random*NCITIES)
  y(i):=round(random*NCITIES)
end-do
forall(i,j in CITIES | 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 CITIES | i<j) DIST(j,i):=DIST(i,j)

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

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

!Se establece el modelo matemático
!-------------------------------------------------------------------------------------------------------------
TotalDist:= sum(i,j in CITIES | i<>j) DIST(i,j)*assig(i,j)
!Sujeto a:
forall(i in CITIES) sum(j in CITIES | i<>j) assig(j,i) = 1  !Entramos a caba ciudad una única vez
forall(i in CITIES) sum(j in CITIES | i<>j) assig(i,j) = 1 !Salimos de cada ciudad una única vez
forall(i,j in CITIES | i<>j) assig(i,j) is_binary !La variable asignación la forzamos a binaria

!Resolvemos el problema y obtenemos los tiempos de ejecución
!-------------------------------------------------------------------------------------------------------------
writeln('NCITIES Seed TotalDistance Time elapsed')
now1:= datetime(SYS_NOW) !Tiempo de inicio
mips := 1;
minimize(TotalDist)
break_subtour
now2:= datetime(SYS_NOW) !Tiempo de finalización

writeln(NCITIES," ", seed, " ",getobjval, " ",  now2-now1)
draw

!FUNCIONES Y PROCEMIENTOS
!=============================================================================================================
!Procedimiento para la eliminación de subciclos
!-------------------------------------------------------------------------------------------------------------
procedure break_subtour
declarations
TOUR,SMALLEST,ALLCITIES: set of integer
end-declarations

forall(i in CITIES)
NEXTC(i):= integer(round(getsol(sum(j in CITIES) j*assig(i,j) )))
! Get (sub)tour containing city 1
TOUR:={}
first:=1
repeat
TOUR+={first}
first:=NEXTC(first)
until first=1
size:=getsize(TOUR)

! Encuentra el subtour más pequeño
if size < NCITIES then
```

```
SMALLEST:=TOUR
if size>2 then
ALLCITIES:=TOUR
forall(i in CITIES) do
if(i not in ALLCITIES) then
TOUR:={}
first:=i
repeat
TOUR+={first}
first:=NEXTC(first)
until first=i
ALLCITIES+=TOUR
if getsize(TOUR)<size then
SMALLEST:=TOUR
size:=getsize(SMALLEST)
end-if
if size=2 then
break
end-if
end-if
end-do
end-if
! Añade una restricción para romper un subtour
        sum(i,j in SMALLEST | i<>j ) assig(i,j) <= getsize(SMALLEST) - 1
        mips := mips+1
draw
! Vuelve a resolver el problema
minimize(TotalDist)
        ! Llama de forma recursiva al procedimiento
break_subtour
end-if
end-procedure

!Procedimiento para dibujar el grafo
!-------------------------------------------------------------------------------------------------------
procedure draw
IVEerase
    IVEzoom(-5,-5,NCITIES+5,NCITIES+5)
    points:=IVEaddplot("cities",IVE_RED)
    roads:=IVEaddplot("roads",IVE_BLUE)

forall(i in 1..NCITIES) do
IVEdrawlabel(points,x(i),y(i),""+i)
end-do
!draw links
forall(i,j in 1..NCITIES) do
     if getsol(assig(i,j)) = 1 then
       IVEdrawarrow(roads,x(i),y(i),x(j),y(j))
     end-if
end-do
end-procedure

end-model
```

A.2. Programa Mosel con variables de potencial

```
(!*****************************************************************************************************
TSP: resolución del problema del TSP formulación con variables de potencial
*****************************************************************************************************!)

!Se cargan las librerías y se le da nombre al modelo
!-----------------------------------------------------------------------------------------------------
model "TSP potencial"
uses "mmxprs","mmive","mmsystem"

!Se advierte de la existencia de determinadas funciones y procedimientos que se encuentran más adelante
!-----------------------------------------------------------------------------------------------------
forward procedure draw

declarations
NCITIES = 25
seed = 1

x: array(1..NCITIES) of integer
y: array(1..NCITIES) of integer

CITIES = 1..NCITIES ! Cities
DIST:  array(CITIES,CITIES) of integer
assig: array(CITIES,CITIES) of mpvar
    level: array(CITIES) of mpvar
now1, now2: datetime
end-declarations

!Se establecen los parámetros de forma aleatoria
!-----------------------------------------------------------------------------------------------------
setrandseed(seed)
forall(i in CITIES) do
  x(i):=round(random*NCITIES)
  y(i):=round(random*NCITIES)
end-do
forall(i,j in CITIES | 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 CITIES | i<j) DIST(j,i):=DIST(i,j)

!MODELO
!=====================================================================================================
writeln("Resolvemos el TSP con n=", NCITIES," ciudades.")


!Se establece el modelo matemático
!-----------------------------------------------------------------------------------------------------
TotalDist:= sum(i,j in CITIES | i<>j) DIST(i,j)*assig(i,j)
!Sujeto a:
forall(i in CITIES) sum(j in CITIES | i<>j) assig(j,i) = 1  !Entramos a caba ciudad una única vez
forall(i in CITIES) sum(j in CITIES | i<>j) assig(i,j) = 1 !Salimos de cada ciudad una única vez
 !Elimina los subciclos
forall(s,t in 2..NCITIES | s<>t)
      level(t) >= level(s) + assig(s,t) - (NCITIES-2) * (1 - assig(s,t)) + (NCITIES-3)*assig(t,s)      ! + (
forall(i,j in CITIES | i<>j) assig(i,j) is_binary !La variable asignación la forzamos a binaria
```

```
!Resolvemos el problema y obtenemos los tiempos de ejecución
!-------------------------------------------------------------------------------------------------------
writeln('NCITIES Seed TotalDistance Time elapsed')
now1:= datetime(SYS_NOW) !Tiempo de inicio
mips := 1;
minimize(TotalDist)
now2:= datetime(SYS_NOW) !Tiempo de finalización

writeln(NCITIES," ", seed, " ",getobjval, " ",  now2-now1)
draw
!FUNCIONES Y PROCEMIENTOS
!=======================================================================================================
!Procedimiento para dibujar el grafo
!-------------------------------------------------------------------------------------------------------
procedure draw
    IVEerase
    IVEzoom(-5,-5,NCITIES+5,NCITIES+5)
    points:=IVEaddplot("cities",IVE_RED)
    roads:=IVEaddplot("roads",IVE_BLUE)

forall(i in 1..NCITIES) do
IVEdrawlabel(points,x(i),y(i),""+i)
end-do
!draw links
forall(i,j in 1..NCITIES) do
     if getsol(assig(i,j)) = 1 then
       IVEdrawline(roads,x(i),y(i),x(j),y(j))
     end-if
end-do
end-procedure
end-model
```

A.3. Programa Mosel con variables de flujo

```
(!*******************************************************************************************************
TSP: resolución del problema del TSP formulación con variables de flujo
*******************************************************************************************************!)
!Se cargan las librerías y se le da nombre al modelo
!-------------------------------------------------------------------------------------------------------
model "TSP flow"
uses "mmxprs","mmive","mmsystem"

!Se advierte de la existencia de determinadas funciones y procedimientos que se encuentran más adelante
!-------------------------------------------------------------------------------------------------------
forward procedure draw

declarations
NCITIES = 25
seed = 1
CITIES = 1..NCITIES
x: array(1..NCITIES) of integer
y: array(1..NCITIES) of integer
```

```
DIST:  array(CITIES,CITIES) of integer ! Distance between cities
assig:   array(CITIES,CITIES) of mpvar ! 1 if we go directly from i to j
flow :   array(CITIES,CITIES) of mpvar
end-declarations

!Se establecen los parámetros de forma aleatoria
!-------------------------------------------------------------------------------------------------------
setrandseed(seed)
forall(i in CITIES) do
  x(i):=round(random*NCITIES)
  y(i):=round(random*NCITIES)
end-do
forall(i,j in CITIES | 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 CITIES | i<j) DIST(j,i):=DIST(i,j)

!MODELO
!=======================================================================================================
writeln("Resolvemos el TSP con n=", NCITIES," ciudades.")

!Se establece el modelo matemático
!-------------------------------------------------------------------------------------------------------
TotalDist:= sum(i,j in CITIES | i<>j) DIST(i,j)*assig(i,j)
!Sujeto a:
forall(i in CITIES) sum(j in CITIES | i<>j) assig(j,i) = 1  !Entramos a caba ciudad una única vez
forall(i in CITIES) sum(j in CITIES | i<>j) assig(i,j) = 1 !Salimos de cada ciudad una única vez
 !Elimina los subciclos
forall(i in CITIES | i<>1) sum(j in CITIES | j<>i) (flow(j,i) - flow(i,j)) = 1
forall(i in 2..NCITIES ) flow(1,i) = (NCITIES-1) * assig(1,i)
forall(i in 2..NCITIES ) flow(i,1) = 0
forall(i in 2..NCITIES , j in 2..NCITIES | i<>j ) flow(i,j)<= (NCITIES-2) * assig(i,j)
forall(i,j in CITIES | i<>j) assig(i,j) is_binary !La variable asignación la forzamos a binaria

!Resolvemos el problema y obtenemos los tiempos de ejecución
!-------------------------------------------------------------------------------------------------------
writeln('NCITIES Seed TotalDistance Time elapsed')
now1:= datetime(SYS_NOW) !Tiempo de inicio
mips := 1;
minimize(TotalDist)
now2:= datetime(SYS_NOW) !Tiempo de finalización
writeln(NCITIES," ", seed, " ",getobjval, " ",  now2-now1)
draw

!FUNCIONES Y PROCEMIENTOS
!=======================================================================================================
!Procedimiento para dibujar el grafo
!-------------------------------------------------------------------------------------------------------
procedure draw
IVEerase
    IVEzoom(-5,-5,NCITIES+5,NCITIES+5)
    points:=IVEaddplot("cities",IVE_RED)
    roads:=IVEaddplot("roads",IVE_BLUE)
forall(i in 1..NCITIES) do
IVEdrawlabel(points,x(i),y(i),""+i)
end-do
```

```
!draw links
forall(i,j in 1..NCITIES) do
     if getsol(assig(i,j)) = 1 then
       IVEdrawline(roads,x(i),y(i),x(j),y(j))
     end-if
end-do
end-procedure
end-model
```

A.4. Programa Mosel con variables de multiflujo

```
(!*****************************************************************************************************
TSP: resolución del problema del TSP formulación con variables de multi flujo
*****************************************************************************************************!)

!Se cargan las librerías y se le da nombre al modelo
!-----------------------------------------------------------------------------------------------------
model "TSP multi-flow"
uses "mmxprs","mmive","mmsystem"

!Se advierte de la existencia de determinadas funciones y procedimientos que se encuentran más adelante
!-----------------------------------------------------------------------------------------------------
forward procedure draw

declarations
NCITIES = 25
seed = 3
x: array(1..NCITIES) of integer
y: array(1..NCITIES) of integer
CITIES = 1..NCITIES ! Cities
DIST:  array(CITIES,CITIES) of integer ! Distance between cities
assig:   array(CITIES,CITIES) of mpvar ! 1 if we go directly from i to j
flow :   array(CITIES,CITIES,CITIES) of mpvar
now1, now2: datetime
end-declarations

!Se establecen los parámetros de forma aleatoria
!-----------------------------------------------------------------------------------------------------
setrandseed(seed)
forall(i in CITIES) do
  x(i):=round(random*NCITIES)
  y(i):=round(random*NCITIES)
end-do
forall(i,j in CITIES | 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 CITIES | i<j) DIST(j,i):=DIST(i,j)

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

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

!Se establece el modelo matemático
!-----------------------------------------------------------------------------------------------------
```

```
TotalDist:= sum(i,j in CITIES | i<>j) DIST(i,j)*assig(i,j)
!Sujeto a:
forall(i in CITIES) sum(j in CITIES | i<>j) assig(j,i) = 1  !Entramos a caba ciudad una única vez
forall(i in CITIES) sum(j in CITIES | i<>j) assig(i,j) = 1 !Salimos de cada ciudad una única vez
 !Elimina los subciclos
forall(k in 2..NCITIES ) do
    sum(j in CITIES | j<>1) flow(1,j,k) = 1
    forall(j in CITIES | j<>1 ) flow(j,1,k) = 0
    sum(j in CITIES | j<>k) flow(j,k,k) = 1
    forall(j in CITIES | j<>k ) flow(k,j,k) = 0
    forall(i in CITIES | i<>k and i<>1) sum(j in CITIES | j<>i) (flow(j,i,k) - flow(i,j,k)) = 0
    forall(i,j in CITIES | i<>j ) flow(i,j,k) <= assig(i,j)
end-do
forall(i,j in CITIES | i<>j) assig(i,j) is_binary !La variable asignación la forzamos a binaria

!Resolvemos el problema y obtenemos los tiempos de ejecución
!-------------------------------------------------------------------------------------------------------
writeln('NCITIES Seed TotalDistance Time elapsed')
now1:= datetime(SYS_NOW) !Tiempo de inicio
mips := 1;
minimize(TotalDist)
now2:= datetime(SYS_NOW) !Tiempo de finalización

writeln(NCITIES," ", seed, " ",getobjval, " ",  now2-now1)
draw

!FUNCIONES Y PROCEMIENTOS
!=======================================================================================================
!Procedimiento para dibujar el grafo
!-------------------------------------------------------------------------------------------------------
procedure draw
IVEerase
    IVEzoom(-5,-5,NCITIES+5,NCITIES+5)
    points:=IVEaddplot("cities",IVE_RED)
    roads:=IVEaddplot("roads",IVE_BLUE)

forall(i in 1..NCITIES) do
IVEdrawlabel(points,x(i),y(i),""+i)
end-do
!draw links
forall(i,j in 1..NCITIES) do
     if getsol(assig(i,j)) = 1 then
       IVEdrawline(roads,x(i),y(i),x(j),y(j))
     end-if
end-do
end-procedure
end-model
```

A.5. Programa Mosel con ramificación y corte

```
(!*****************************************************************************************************
TSP: Resolución del TSP, a partir del método exacto de resolución llamado ramificación y corte. Para encontr
Se hará uso de funciones de llamada recursiva, a través de la sentencia "setcallback".
*****************************************************************************************************!)
!Se cargan las librerías y se le da nombre al modelo
!-----------------------------------------------------------------------------------------------------
model "Tsp subtour elimination"
uses "mmxprs","mmive", "mmsystem"

!Se modifican los parámetros utilizados por el resolutor
!-----------------------------------------------------------------------------------------------------
setparam("XPRS_HEURSTRATEGY", 0)  !Estrategia heurística, 0 indicaría no heurístico
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 más adelante
!-----------------------------------------------------------------------------------------------------
forward procedure draw
forward public function cb_node:boolean
forward public procedure cb_entera(isheur:boolean,cutoff:real)

declarations
NCITIES = 125
root    = 1
CITIES  = 1..NCITIES
epsilon = 0.0001
seed = 5
x: array(CITIES) of integer
y: array(CITIES) of integer
DIST:  array(CITIES,CITIES) of integer
addcut_in_cb_entera: boolean
cut: linctr
now1, now2: datetime
assig: array(CITIES,CITIES) of mpvar
end-declarations

addcut_in_cb_entera:= false

!Se establecen los parámetros de forma aleatoria
!-----------------------------------------------------------------------------------------------------
setrandseed(seed)
forall(i in CITIES) do
  x(i):=round(random*NCITIES)
  y(i):=round(random*NCITIES)
end-do
forall(i,j in CITIES | 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 CITIES | i<j) DIST(j,i):=DIST(i,j)

!MODELO
!=====================================================================================================
writeln("Resolvemos el TSP con n=", NCITIES," ciudades.")
```

```
!Se establece el modelo matemático
!-------------------------------------------------------------------------------------------------------
TotalDist:= sum(i,j in CITIES | i<>j) DIST(i,j)*assig(i,j)
!Sujeto a:
forall(i in CITIES) sum(j in CITIES | i<>j) assig(j,i) = 1  !Entramos a caba ciudad una única vez
forall(i in CITIES) sum(j in CITIES | i<>j) assig(i,j) = 1 !Salimos de cada ciudad una única vez
forall(i,j in CITIES | i<j) assig(i,j)+assig(j,i) <= 1 !Elimina los subciclos de tamaño 2
!forall(i,j in CITIES | i<>j) assig(i,j) is_binary !La variable asignación la forzamos a binaria

!Resolvemos el problema y obtenemos los tiempos de ejecución
!-------------------------------------------------------------------------------------------------------
writeln('NCITIES Seed TotalDistance Time elapsed')
now1:= datetime(SYS_NOW) !Tiempo de inicio

!Se genera el procedimiento de ramificación y corte, a través de funciones recursivas
!-------------------------------------------------------------------------------------------------------
setcallback(XPRS_CB_CUTMGR, "cb_node")
setcallback(XPRS_CB_PREINTSOL, "cb_entera")
minimize(TotalDist)
now2:= datetime(SYS_NOW) !Tiempo de finalización
writeln(NCITIES," ", seed, " ",getobjval, " ",  now2-now1)
draw

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

!Procedimiento llamado cuando es encontrada una solución entera mediante heurístico o ramificación y acotaci
!Si la solución tiene subciclos es desechada, siendo no válida la cota superior asociada.
!-------------------------------------------------------------------------------------------------------
public procedure cb_entera(isheur:boolean,cutoff:real)
declarations
TOUR,SMALLEST,ALLCITIES: set of integer
node: integer
end-declarations

!    writeln("Dentro de ENTERA: heurístico=",isheur,", cota=",cutoff,", profundidad= ",getparam("XPRS_NODEDE
!     ", nodo=",getparam("XPRS_NODES"),", valor objetivo= ", getparam("XPRS_LPOBJVAL"))
forall(i in CITIES)  !Para cada nodo se indentifica quién es su hijo
NEXTC(i):= ceil( getsol( sum(j in CITIES) j*assig(i,j) )-epsilon )
!Busca un subciclo
TOUR:={}
first:=root
repeat
TOUR+={first}
first:=NEXTC(first)
until first=root
size:=getsize(TOUR)

!Si hay subciclos identifica el menor conjunto de nodos generatices de uno de ellos
if size < NCITIES then
rejectintsol
        if (addcut_in_cb_entera) then
SMALLEST:=TOUR
```

```
if size>3 then
ALLCITIES:=TOUR
forall(i in CITIES) do
if(i not in ALLCITIES) then
TOUR:={}
first:=i
repeat
TOUR+={first}
first:=NEXTC(first)
until first=i
ALLCITIES+=TOUR
if getsize(TOUR)<size then
SMALLEST:=TOUR
size:=getsize(SMALLEST)
end-if
if size=3 then
break
end-if
end-if
end-do
end-if
!Añade el corte, si procede
        cut:=sum(i,j in SMALLEST | i<>j ) assig(i,j) - getsize(SMALLEST) + 1
        addcut(1, CT_LEQ, cut)
    end-if
      end-if
end-procedure

!Función que indentifica cortes violados, mediante un problema de flujo máximo.
!Este genera una restricción que será añadida al nodo actual y a sus hijos.
!Se procederá a ramificar cuando no sean encontrados flujos máximos menores que 1, desde la raíz al resto.
!-------------------------------------------------------------------------------------------------------
public function cb_node:boolean
declarations
Capacity,Flow:array (CITIES,CITIES) of real
first,last, SINK,SOURCE :integer
TOUR  : array(CITIES) of integer
inside: array(CITIES) of boolean
maxflow: mpproblem
amount:real
flow: array(CITIES,CITIES) of mpvar
end-declarations

addcut_in_cb_entera:= true

!Creamos un grafo con cargas, las cuales se corresponden con la solución parcial
forall(n,m in CITIES|n<>m) Capacity(n,m) := getsol(assig(n,m))
forall(i,j in CITIES|i<>j and Capacity(i,j)<epsilon ) Capacity(i,j) := 0

returned:=false

  !Resolvemos el problema de flujo máximo
SOURCE:=root
forall(j in CITIES|j<>SOURCE)do
```

```
SINK:=j
with maxflow do
reset( maxflow )
forall(n,m in CITIES |n<>m and Capacity(n,m)>epsilon )  create(flow(n,m))
  forall(n in CITIES | n<>SOURCE and n<>SINK)
         sum(m in CITIES|m<>n and exists(flow(m,n))) flow(m,n) = sum(m in CITIES|m<>n and exists(flow(n,m)))
  forall(n in CITIES | n<>SOURCE and exists(flow(n,SOURCE)) ) flow(n,SOURCE) = 0
  forall(n,m in CITIES | n<>m and exists(flow(n,m)) )         flow(n,m)<=Capacity(n,m)
  maximize( sum(n in CITIES|n<>SOURCE and exists(flow(SOURCE,n))) flow(SOURCE,n) )
  amount := getobjval
  forall (n,m in CITIES | n<>m) Flow (n,m):= getsol(flow(n,m))
end-do

!Identificamos si el flujo es menor que uno, para posteriormente encontrar una restricción de subciclos viol
  if amount < 1-epsilon then
forall(i in CITIES) inside(i):=false
TOUR(1):= SOURCE
inside(SOURCE):= TRUE
last :=1
while (last>0) do
u := TOUR(last)
last := last-1
forall ( v in CITIES | 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

!Añadimos el corte oportuno
cut  := ( sum(i,k in CITIES | inside(i)=true and inside(k)=false) assig(i,k) ) -1
addcut(1, CT_GEQ, cut)

! writeln("Corte añadido: (profundidad= ",getparam("XPRS_NODEDEPTH"),", nodo= ",getparam("XPRS_NODES"),
! ", valor objetivo= ", getparam("XPRS_LPOBJVAL"), ")")

returned:=true
break
  end-if
end-do
! if returned = false then
! writeln("Ramificando")
! end-if
end-function

!Procedimiento para dibujar el grafo
!-------------------------------------------------------------------------------------------------
procedure draw
declarations
points,roads1,roads2,roads3,roads4:integer
end-declarations
```

```
IVEerase
    IVEzoom(-5,-5,NCITIES+5,NCITIES+5)
    first:=IVEaddplot("root",IVE_MAGENTA)
    points:=IVEaddplot("Cities",IVE_RED)
    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,j in CITIES|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

IVEdrawlabel(first,x(root),y(root),""+root)
    forall(i in 2..NCITIES) IVEdrawlabel(points,x(i),y(i),""+i)
end-procedure
end-model
```

Bibliografía

[1] David L. Applegate · Robert E.Bixby · Vasek Chvátal · William J. Cook. *The Traveling Salesman Problem*. Princeton University Press, 2006.

[2] FICO. *Xpress-Mosel (User Guide)*, March 2012.

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

[4] G.B. Dantzig · D.R.Fulkerson · S. Johnson. Solution of a large scale traveling salesman problem. 1954.

[5] Robert Fourer · David M. Gay · Brian W. Kernighan. *AMPL: A Modeling Languague for Mathematical Programming*. Duxbury Press · Brooks/Code Publishing Company, 2002.

[6] Aida Calviño Martínez. Cooperación en los problemas del viajante (tsp) y de rutas de vehículos (vrp): una panorámica. 2011.

[7] H. Mittelmann. Mixed integer linear programming benchmark, Marzo 2014.

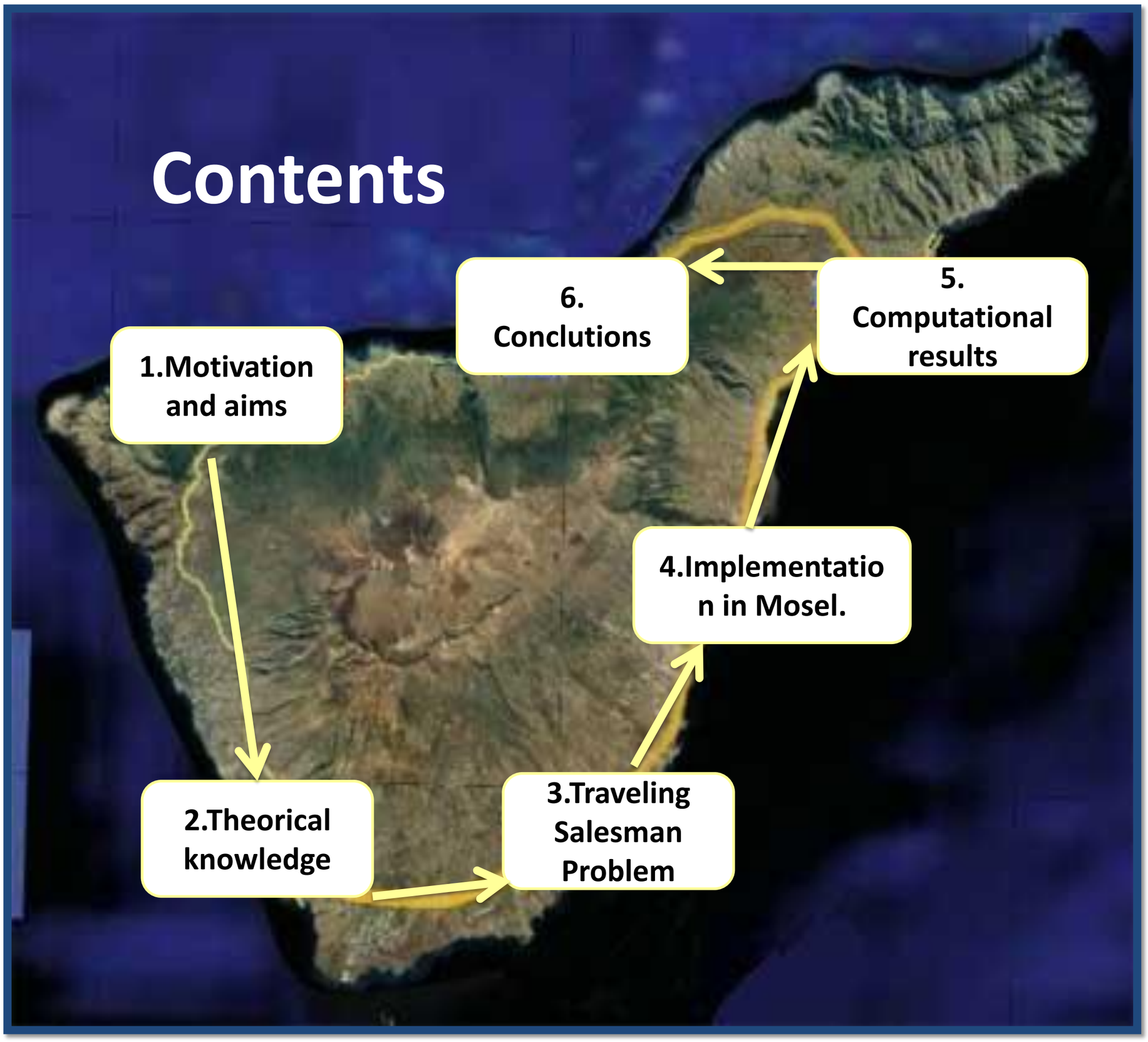
[8] Hipólito Hernández Pérez. *Procedimientos exactos y heurísticos para resolver problemas de rutas con recogida y entrega de mercancía*. Servicio de publicaciones Universidad de La Laguna, 2004-2005.

[9] Adam N. Letchford · Juan José Salzar. Projection results for vehicle routing. 2004.

[10] Xpress Team. *Xpress-Optimizer Reference manual*. Fair Isaac Corporation, 2009.

TRAVELING SALESMAN PROBLEM (TSP)

Exact resolving methods



María Victoria García Travieso
July 21st 2014
Facultad de Matemáticas
ULL

In many sectors of society to **optimize** resources and benefits is necessary, or even essential. That is why the need to compare several mathematical models to solve certain problems of combinatorial programming.

In particular, the classic problem known as Traveling Salesman Problem (**TSP**) can be related with aspect of the everyday life which we are not able to image. The propose of this project is to **compare** some mathematical models solving this problem, where are including exact models with: subtour constraints, flow variables, multiflow variables, potencial variables and a new way to implement branch and cut using the resource "**setcallback**" given by Mosel (programming language).