FACULTADE DE MATEMÁTICAS

Trabajo Fin de Grado

Optimización combinatoria y algoritmos heurísticos

Diego González González

2024/2025

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRADO DE MATEMÁTICAS

Trabajo Fin de Grado

Optimización combinatoria y algoritmos heurísticos

Diego González González

07/2025

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Trabajo propuesto

Área de Conocimiento: Estadística e Investigación Operativa
Título: Optimización combinatoria y algoritmos heurísticos
Breve descripción del contenido
El objetivo de este trabajo es efectuar una introducción al estudio de la optimización combinatoria. Para ello, primeramente y siguiendo las referencias [2] y [14], se revisarán algunos de los problemas clásicos de dicha optimización combinatoria, prestando atención a su formulación, resultados teóricos de relevancia y algoritmos con un buen comportamiento. Además, en esta disciplina resulta de particular interés el análisis de los denominados algoritmos heurísticos. Dichos algoritmos son apreciados desde campos diversos como la logística o la inteligencia artificial, por lo cual realizaremos su estudio, siguiendo [1] y desde un enfoque práctico y aplicado.

Índice

Resumen VI

Introducción IX

1. Preliminares **1**

1.1. El problema de programación lineal 1

1.2. Álgebra lineal y análisis convexo 3

1.3. Dualidad 5

1.3.1. El problema de programación lineal dual 5

1.3.2. Relación primal-dual 5

1.4. Clases de complejidad 6

1.5. Búsqueda local 8

1.6. Teoría de grafos 11

1.6.1. Una revisión de conceptos básicos de grafos 11

2. El problema de flujo en redes **13**

2.1. Teorema de máximo flujo mínimo corte 15

3. El problema de la mochila **21**

3.1. Algunas modificaciones del problema de la mochila 21

3.2. Resolución del problema de la mochila 25

4. El problema del viajante de comercio **29**

4.1. Algoritmos de construcción de tours 33

5. Simulación del TSP **37**

5.1. Definición del problema 37

5.1.1. Galicia 38

5.1.2. Noroeste de España 41

5.1.3. Norte de España 43

I. Código del algoritmo *Clarke-Wright* **55**

II. 10 ejecuciones de los algoritmos de la librería *TSP* **59**

III.Código y solución del templado simulado **61**

IV.Archivo para Concorde **69**

Bibliografía **71**

Resumen

El trabajo principalmente consistirá en el análisis exhaustivo de problemas de optimización combinatoria, en particular, de tres de ellos: problema de flujo en redes, problema de la mochila y el problema del viajante de comercio. A este último le acompañará una simulación práctica aplicada a la realidad, de modo que se analizará la eficiencia de sus principales heurísticos.

Abstract

The work will primarily consist of an in-depth analysis of combinatorial optimization problems, particularly three of them: the network flow problem, the knapsack problem, and the traveling salesman problem. The latter will include a practical simulation applied to real-world scenarios, allowing for an analysis of the efficiency of its main heuristics.

Introducción

La programación matemática, como subcampo de la investigación operativa, es un área de las matemáticas que procura crear y resolver modelos matemáticos de optimización. Los problemas de óptimización surgen de manera natural en distintos ámbitos del ser humano como la ingeniería, la economía, la logística... Una de las subsecciones más importantes es la optimización combinatoria, que se vale de la programación para encontrar soluciones óptimas dentro de un conjunto discreto de posibles soluciones con la presencia de ciertas restricciones. El desarrollo histórico de la programación matemática, por tanto, será de gran utilidad para visualizar la evolución técnica y teórica para la resolución de distintos problemas de optimización y, en particular, los de optimización combinatoria.

Uno de los primeros problemas conocidos que se valieron de la programación como herramienta de resolución es el problema del viajante. Con el desarrollo del comercio a gran escala, cada vez se busca más la creación de técnicas para resolver este tipo de problemas logísticos.

Otro de los problemas que impulsaron a la programación es el problema de la dieta, propuesto en la década de 1930 por los Estados Unidos. Se trataba de procurar hallar la cantidad y variedad de alimentos necesaria para aportar a los ciudadanos los nutrientes imprescindibles pero con el menor costo posible. Un problema el cual pretende la óptimización de una función objetivo en presencia de restricciones lineales, es decir, las bases de la programación lineal. Debido también a la preocupación por cuestiones como la asignación de trabajos, provocaron una necesidad de modelos matemáticos precisos pero computacionalmente eficientes.

Un gran salto en el ámbito fue propiciado por la Segunda Guerra Mundial. Este fue un período en el cual se llevaba a cabo una guerra total: todos los eslavones de la sociedad estaban implicados. La necesidad de racionalizar los suministros y concretar objetivos militares, demandó la innovación en modelos cuantitativos. En este contexto, George Dantzig, que trabajaba como asesor matemático de la Fuerza Aérea de los Estados Unidos, en 1947 desarrolló el método simplex; una herramienta básica en la programación lineal actual. Esto, junto al avance de la informática, permitió lo que era impensable hasta ese momento: poder trabajar con cientos y hasta miles de variables a la vez.

A lo largo del tiempo se llegó a la conclusión de que no siempre ciertas realidades podían representarse en términos continuos o lineales. Por lo tanto, con la necesidad de tratar variables enteras o discretas, florece la optimización combinatoria. En esta área destacan problemas célebres como el ya mencionado problema del viajante de comercio, que se basa en procurar encontrar la ruta más corta entre un conjunto de ciudades, pasando por todas ellas y volviendo al punto de partida.

El desarrollo de la informática y de la capacidad computacional ha sido de gran importancia en el ámbito de la optimización combinatoria. Esto se debe a que cuanto mayor potencia computacional, mayor velocidad de resolución de problemas cada vez más sofisticados con un mayor número de variables. Por ello, desde el nacimiento de los primeros ordenadores hasta los de última generación, han permitido la creación de algoritmos cada vez más complejos, así como la innovación en ramas como la heurística y la metaheurística que permiten ofrecer soluciones aproximadas en un tiempo razonable para problemas cuya solución exacta resulta incalculable.

En síntesis, la optimización con la ayuda de las herramientas de la programación matemática y las mejoras informáticas y computacionales, resultan indispensables para la resolución de problemas sociales, económicos y tecnológicos de la actualidad. De hecho, hoy en día la optimización combinatoria tiene gran relevancia debido al florecimiento de la inteligencia artificial.

En nuestro caso específico, daremos un paso desde los problemas de programación lineal más básicos a los problemas más indispensables de la optimización combinatoria. Para ello, iniciaremos el trabajo con un breve recordatorio de qué son, sus principales características y la importancia de las clases de complejidad. A continuación, nos centraremos en tres de los problemas de combinatoria más relevantes: el problema del flujo, el problema de la mochila y el problema del viajante de comercio. Respecto a ellos, estudiaremos sus principales heurísticos asociados y analizaremos su eficiencia. Por último, fabricaremos un ejemplo aplicado a la realidad referido al TSP, en el que nos valdremos de la ayuda de los lenguajes de programación **R** y **AMPL** para su simulación y resolución. De esta forma, analizaremos el buen comportamiento de sus principales algoritmos.

Capítulo 1

Preliminares

El objetivo principal de este trabajo es abordar distintos problemas de optimización combinatoria ilustrados por medio de aplicaciones prácticas. Por esta razón, previamente debemos introducir alguna información básica sobre los problemas de programación lineal, al mismo tiempo que introducimos ciertos resultados fundamentales.

1.1. El problema de programación lineal

El problema de programación lineal, tal y como se desarrolla en [2], se basa en encontrar la solución que minimice o maximice una función lineal en presencia de igualdades lineales o restricciones de desigualdad. Esto supone la base de la resolución de problemas más complejos como es el caso de la optimización combinatoria.

Vamos a comenzar formulando un tipo particular de problema de programación lineal que como veremos, cualquier otro problema de programación lineal podrá manipularse hasta obtener esta forma.

Construcción del modelo

Definamos como $z = c_1x_1 + c_2x_2 + \cdots + c_nx_n$ a la función objetivo. Los valores $c_1, c_2, ..., c_n$ se conocen como *coeficientes de coste* mientras que $x_1, x_2, ..., x_n$ son las *variables de decisión* que queremos hallar. Dado que buscamos minimizar la función objetivo, formalmente será expresada del siguiente modo.

$$\text{Minimizar} \quad z = c_1x_1 + c_2x_2 + \cdots + c_nx_n.$$

A este objetivo añadiremos un conjunto de restricciones sobre los valores de las variables. A

continuación, las inecuaciones $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$ para $i = 1, \ldots, m$ denotan la i-ésima restricción en la cual los coeficientes a_{ij} se llaman *coeficientes tecnológicos.*

$$\sum_{j=1}^{n} a_{ij}x_j \geq b_i, \quad \text{para } i = 1, 2, \ldots, m$$

$$x_j \geq 0, \quad \text{para } j = 1, 2, \ldots, n.$$

De esta forma, se da lugar a la matriz de restricciones $\mathbf{A}$.

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}.$$

Definimos como *vector lado derecho* al vector columna formado por los coeficientes b_i, que representa ciertas condiciones mínimas requeridas. Un problema representado de este modo se dice que está en *forma canónica.* Mientras que también existe otra formulación muy empleada, la *forma estándar*, que en lugar de representar las restricciones como desigualdades lo hace mediante igualdades.

Un conjunto de valores de las variables $x_1, \ldots, x_n$ satisfaciendo todas las restricciones se denomina *punto factible.* El conjunto formado por dichos puntos construye la llamada *región factible* o *espacio factible.*

Por tanto, el problema de programación lineal se basa en encontrar entre todos los puntos factibles aquellos que minimicen (o maximicen) la función objetivo, es decir, en encontrar las soluciones óptimas.

Manipulación del problema

En un problema de programación lineal sabemos que están presentes restricciones a modo de desigualdades o igualdades. El problema puede ser transformado de una forma a otra equivalente modificando dicho problema mediante unas manipulaciones simples.

1- **Inecuaciones y ecuaciones**. Una inecuación puede ser convertida fácilmente en una ecuación. Para ilustrarlo, consideremos la restricción dada por $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$. Se puede modificar en forma de ecuación añadiendo una *variable de holgura* s_i no negativa de forma que nos queda: $\sum_{j=1}^{n} a_{ij}x_j - s_i = b_i$ y $s_i \geq 0$. Equivalentemente en el caso $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ tenemos:

$\sum_{j=1}^{n} a_{ij}x_j + s_i = b_i$ y $s_i \geq 0$. Además, una ecuación de la forma $\sum_{j=1}^{n} a_{ij}x_j = b_i$ puede ser transformada en dos inecuaciones: $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$ y $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$.

2- **Problemas de maximización y minimización**. Otra clase de manipulación que podemos hacer es tratar de convertir un problema de maximización en uno de minimización y viceversa. Cabe destacar que para una región dada,

$$\text{máx} \sum_{j=1}^{n} c_j x_j = -\text{mín} \sum_{j=1}^{n} -c_j x_j.$$

Por lo cual, un problema de un tipo puede ser transformado en un problema del otro tipo únicamente multiplicando la función objetivo por -1.

1.2. Álgebra lineal y análisis convexo

Nos valemos de la información de [2] para el desgranado de esta sección.

Definición 1.1. Sean **a** y **b** dos vectores de $\mathbb{R}^n$. El **producto interior** se define como:

$$\mathbf{ab} = a_1b_1 + a_2b_2 + \cdots + a_{n-1}b_{n-1} + a_nb_n = \sum_{i=1}^{n} a_ib_i.$$

Definición 1.2. La **norma** de un vector **a** de $\mathbb{R}^n$ (consideraremos la norma euclídea) viene dada por:

$$||a|| = \sqrt{\sum_{i=1}^{n} a_i^2}.$$

Definición 1.3. Sea un conjunto A de $\mathbb{R}^n$. Se dice que es *convexo* si dados dos puntos de A, x_1 y x_2, entonces $\lambda x_1 + (1-\lambda)x_2 \in A$, para cada $\lambda \in [0,1]$. Nótese que $\lambda x_1 + (1-\lambda)x_2$, para $\lambda \in [0,1]$, representa un punto del segmento que une a x_1 con x_2. A cualquier punto de este segmento se dice que es una combinación convexa de x_1 y x_2.

Observación 1.4. La convexidad puede ser interpretada gráficamente de la siguiente forma. Para cada par de puntos x_1, x_2 de A, el segmento que une a estos puntos debe estar contenido en A. Los siguientes son ejemplos de un conjunto convexo y otro no convexo, respectivamente.

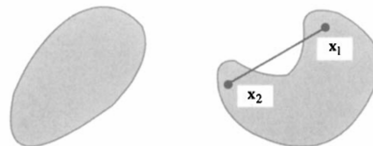

Figura 1.1: Convexidad.

Definición 1.5. Dada una combinación convexa como la introducida en la definición anterior, si $\lambda \in (0,1)$, se le llama *estricta*.

Ejemplo 1.6. Los conjuntos siguientes verifican la propiedad de convexidad.

1. $\{(x_1, x_2) \in \mathbb{R}^2 : (x_1)^2 + (x_2)^2 \leq 1\}$.

2. $\{x \in \mathbb{R}^n : \mathbf{A}x = \mathbf{b}\}$, donde $\mathbf{A}$ es una matriz $m \times n$ y $\mathbf{b}$ es un vector m-dimensional.

3. $\{x \in \mathbb{R}^n : \mathbf{A}x \leq \mathbf{b}\}$, donde $\mathbf{A}$ es una matriz $m \times n$ y $\mathbf{b}$ es un vector m-dimensional.

4. $\left\{ \mathbf{x} \in \mathbb{R}^3 : \mathbf{x} = \lambda_1 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \lambda_2 \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} + \lambda_3 \begin{pmatrix} -1 \\ 2 \\ -3 \end{pmatrix}, \quad \lambda_1 + \lambda_2 + \lambda_3 = 1, \quad \lambda_1, \lambda_2, \lambda_3 \geq 0 \right\}$.

5. $\{x \in \mathbb{R}^n : \mathbf{p}^T x \leq \alpha\}$, donde $\mathbf{p} \in \mathbb{R}^n, \alpha \in \mathbb{R}$.

Definición 1.7. Sea un conjunto convexo A y un punto $x \in A$. Se le llama *punto extremo* de A si no puede ser representado como combinación convexa estricta de dos puntos distintos de A, es decir, si $x = \lambda x_1 + (1-\lambda)x_2$ con $\lambda \in (0,1)$ y $x_1, x_2 \in A$ entonces $x = x_1 = x_2$.

En el ejemplo siguiente, x_2 y x_3 son puntos no extremos mientras que x_1 es un punto extremo.

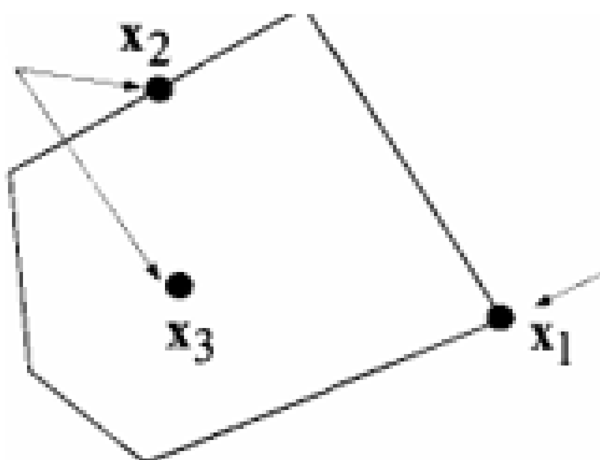

Figura 1.2: Punto extremo.

1.3. Dualidad

Siguiendo [2], para cualquier problema lineal que procuremos resolver, denotado como problema *primal*, existe otro problema lineal asociado, el problema *dual*, que cumple unas propiedades muy importantes. Además este se puede emplear para obtener la solución del problema original y sus variables nos ofrecen información muy valiosa acerca del conjunto de soluciones óptimas del problema original.

1.3.1. El problema de programación lineal dual

Se dice que un problema de programación lineal, (P), está en su *forma canónica* si el objetivo es el de minimizar cx, las restricciones funcionales son de mayor o igual, esto es, $Ax \geq b$, y las variables son no negativas, es decir, $x \geq 0$. Partiendo de este punto, el problema de programación lineal dual, (D), es aquel en el que el objetivo es el de maximizar wb, donde w representa el nuevo vector de variables. Las restricciones funcionales son ahora $wA \leq c$ y las variables son no negativas, esto es, $w \geq 0$.

Cabe destacar que hay exactamente una variable dual para cada restricción del problema primal y exactamente una restricción de menor o igual en el dual por cada variable del primal.

Una representación alternativa es la denominada *forma estándar* de un problema de programación lineal que es como sigue.

$$\begin{array}{lll} \text{(P)} & \min\limits_{x} & cx \\ & \text{sujeto a} & Ax = b \\ & x \geq 0. & \end{array}$$

Entonces, el problema dual tiene la siguiente forma:

$$\begin{array}{lll} \text{(D)} & \max\limits_{w} & wb \\ & \text{sujeto a} & wA \leq c \\ & w \quad \text{sin restricciones de signo.} & \end{array}$$

1.3.2. Relación primal-dual

La definición anterior del problema dual nos lleva a algunas interesantes relaciones entre los problemas de programación lineales dual y primal.

La relación entre valores objetivo

Lema 1.8 (Propiedad de dualidad débil). *Sean un par de problemas de programación lineal duales, llamémosles primal y dual. Supongamos que el primal es un problema de minimización y el dual de maximización. Entonces, el objetivo en cualquier solución factible del dual es menor o igual que el objetivo en cualquier solución factible del primal.*

Demostración. Consideremos la forma canónica de dualidad y sean $\mathbf{x}_0$ y $\mathbf{w}_0$ soluciones factibles del problema primal y dual respectivamente. Multiplicando por $\mathbf{w}_0 \geq 0$ por la izquierda a $\mathbf{Ax}_0 \geq \mathbf{b}$ y por $\mathbf{x}_0 \geq 0$ y la derecha a $\mathbf{w}_0\mathbf{A} \leq \mathbf{c}$ obtenemos:

$$\mathbf{cx}_0 \geq \mathbf{w}_0\mathbf{Ax}_0 \geq \mathbf{w}_0\mathbf{b}$$

de donde se sigue el resultado. □

Como consecuencia inmediata del lema que acabamos de presentar, se deducen fácilmente los siguientes corolarios.

Corolario 1.9. *Consideremos un par de soluciones factibles, una del problema primal y otra del problema dual. Si estas soluciones otorgan el mismo valor a los correspondientes objetivos entonces son necesariamente óptimas.*

Corolario 1.10. *Sea un problema de programación lineal y su problema dual. Si uno de ellos tiene un objetivo no acotado, el otro no puede tener soluciones factibles.*

1.4. Clases de complejidad

En esta sección se revisan algunos conceptos básicos de complejidad, basados en [6], de una forma intuitiva que conviene tener presentes en el resto del trabajo.

Definición 1.11. Dado un problema de decisión, decimos que pertenece a la clase de complejidad **P** si existe un algoritmo que resuelva cualquier ejemplo de dicho problema en tiempo polinomial.

Definición 1.12. Dado un problema de decisión, decimos que pertenece a la clase de complejidad **NP** si existe un algoritmo que puede verificar cualquier solución a un ejemplo en dicha clase en tiempo polinomial.

No debemos confudir la clase **NP** con los problemas "no polinomiales".

Es importante resaltar que $\mathbf{P} \subset \mathbf{NP}$, pues cualquier problema fácil de resolver será fácil de verificar. Dado un ejemplo de una clase de problemas en **P** y una solución dada, puedo verificar

dicha solución en tiempo polinomial sin más que resolver el ejemplo (puedo hacerlo en tiempo polinomial por estar en **P**) y verificar a partir de dicha resolución la solución dada.

La gran pregunta de la teoría de complejidad computacional es si las dos clases que acabamos de definir coinciden. Es decir, si **P**=**NP** o, lo que es igual, si cualquier problema fácil de verificar será también fácil de resolver. Aunque hay un amplio consenso acerca de que ambas clases deben ser distintas, nadie ha conseguido probarlo aún. En caso de que ambas clases fuesen iguales. esto tendría gran afectación en multitud de materias.

En el estudio de las distintas clases de complejidad resulta de especial utilidad transformar unos problemas en otros. Diremos que P_1 se puede reducir a P_2 en tiempo polinomial si cualquier ejemplo en dicha clase se puede transformar en tiempo polinomial en un ejemplo de la clase P_2. Si la clase P_2 pertenece a **P**, entonces también la clase P_1 pertenece a **P**. Dado un elemento de P_1 lo podremos resolver en tiempo polinomial sin más que transformarlo previamente (en tiempo polinomial) en un elemento de P_2.

Definición 1.13. Un problema P perteneciente a la clase **NP** se dice **NP-completo** si cualquier problema perteneciente a la clase **NP** se puede reducir al problema P en tiempo polinomial.

Esta caracterización permite considerar a los problemas **NP-completos** como los más difíciles de entre todos los de la clase **NP**. Por tanto, dado un algoritmo para el problema P_1, puede ser adaptado para resolver en un tiempo extra polinomial cualquier otro problema de la clase **NP**. Entonces, si apareciese algún algoritmo polinomial para resolver un problema **NP-completo**, tendríamos que cualquier problema en **NP** se puede resolver también en tiempo polinomial, lo que implicaría que **P**=**NP**.

Definición 1.14. Un problema P se dice que pertenece a la clase **NP-duro** cuando cualquier problema P' de la clase **NP** se puede reducir a P en un tiempo polinomial.

La diferencia entre las clase **NP-completo** y **NP-duro** es que un problema puede ser **NP-duro** sin pertenecer a la clase **NP**.

El hecho de que no seamos capaces de encontrar algoritmos polinomiales para un determinado problema no quiere decir que debamos olvidarnos de dicho problema. Al fin y al cabo, el criterio del "análisis del peor caso" es solo una regla de decisión, y podríamos haber aplicado muchas otras. De hecho, este criterio se corresponde con un enfoque conocido como *maxmín* en teoría de decisión: dado un conjunto de algoritmos, nuestro decisor mira cuál es el peor caso para cada algoritmo y después se queda con el algoritmo que tenga el mejor peor caso.

1.5. Búsqueda local

La búsqueda local es un procedimiento vastamente empleado para resolver problemas difíciles de optimización, por lo que se indagará en él teniendo en cuenta [1]. Para obtener una heurística de búsqueda local para un problema de optimización, se superpone en las soluciones una estructura de entorno, es decir, para cada solución se especifica un conjunto de "soluciones vecinas". En adición, para la correcta inicialización de tal algoritmo heurístico, es necesaria la generación de una solución inicial, que puede provenir de otro algoritmo o simplemente puede ser calculada por un mecanismo aleatorio. A partir de esta, el heurístico procura encontrar una solución vecina mejor, hasta que se encuentra con una solución óptima local.

Hay que tener en cuenta que los dos problemas principales sobre los heurísticos de búsqueda local son la calidad de las soluciones locales obtenidas y como estas se relacionan realmente con el óptimo global y la complejidad del heurístico de búsqueda local, esto es, como de rápido puede encontrar el óptimo local. Cuanto más generales se escojan las vecindades mejor será la solución, sin embargo, más difícil será calcularla. Por lo que, crear un buen heurístico de búsqueda local atañe a encontrar un buen equilibrio entre dificultad computacional y calidad de solución.

Empíricamente se puede observar que los heurísticos de búsqueda local parece que convergen relativamente rápido. Los problemas cuya vecindad puede ser encontrada en tiempo polinomial, definieron en [12] una nueva clase de complejidad llamada **PLS** *(polynomial-time local search)*. Este suele ser el mínimo requerimiento de los heurísticos de búsqueda local más comunes. La clase **PLS** se posiciona de algún modo entre las clases **P** y **NP** ya definidas previamente.

Dado un problema computacional general Π, que pretende resolver una tarea mediante el uso de un algoritmo, posee un conjunto D_Π de instancias (casos) y para cada instancia $x \in D_\Pi$ hay un conjunto $O_\Pi(x)$ de posibles respuestas aceptables. Un algoritmo resuelve el problema Π si recibe como dato un $x \in D_\Pi$ y saca como respuesta un elemento $y \in O_\Pi(x)$ (en caso de que $O_\Pi(x)$ sea vacío, el algoritmo nos informaría que ningún y existe). Los elementos de entrada (inputs) y de salida (outputs) se representan como palabras de un alfabeto finito, que sin pérdida de generalidad podemos tomar como $\{0,1\}$.

Un ejemplo de problema computacional para ejemplificar algunas de estas cosas puede ser: Π=*Determinar si un número natural dado en binario es par*. Ahora tomamos el alfabeto finito $\Sigma = \{0, 1\}$. A continuación, definimos el conjunto de instancias del problema:

$$D_\Pi = \{x \in \{0,1\}^* \mid x \text{ representa un número natural binario}\}.$$

(Apréciese que con la notación $\{0,1\}^*$ nos referimos al conjunto de cadenas posibles que podemos formar con el 0 y el 1).

El conjunto de respuestas aceptables para cada instancia se describe como sigue:

$$O_\Pi(x) = \begin{cases} \{1\} & \text{si } x \text{ representa un número par} \\ \{0\} & \text{si } x \text{ representa un número impar.} \end{cases}$$

Entonces, un posible algoritmo para resolver el problema lo que haría es tomar como entrada una palabra $x \in D_\Pi$, luego ver si el último bit de x es 0 y por último, devuelve 1 en ese caso y 0 en caso contrario.

Cómo sea el conjunto de posibles salidas aceptables $O_\Pi(x)$ determina la clase de problema computacional a la que nos estamos refiriendo. Por un lado, los *problemas de búsqueda* verifican que $O_\Pi(x)$ puede contener ninguno, uno o algún elemento, mientras que los problemas *locales* son aquellos en los que se cumple que $O_\Pi(x)$ no es nunca el conjunto vacío $\forall x \in D_\Pi$. Otro tipo de problemas destacables son los *funcionales* los cuales verifican que $\forall x$, $|O_\Pi(x)| \leq 1$. Un importante caso especial de problemas funcionales locales son los *problemas de decisión* en los cuales las únicas respuestas son "Sí"(1) o "No"(2).

En lo que se refiere a este trabajo, nos centraremos en una clase particular de problemas de búsqueda llamados **problemas de optimización combinatoria**. Sus principales características son que $\forall x \in D_\Pi$, $\varphi_\Pi(x)$, que se refiere al conjunto de soluciones en este caso; es finito y que $\forall i \in \varphi_\Pi(x)$, tiene asociada un coste $f_\Pi(i, x)$. El problema, dado un x, encuentra una solución óptima $i^* \in \varphi_\Pi(x)$ ya sea un problema de maximización (coste máximo) o un problema de minimización (coste mínimo). Cabe destacar que pueden existir casos en los que convivan varias soluciones óptimas distintas con un mismo coste óptimo.

Mediante la descripción de la llamada *función de vecindad*, definida sobre el conjunto de soluciones, se puede evolucionar de un problema de optimización combinatoria a un **problema de búsqueda local**. Esto significa que, dada una instancia del problema x y una solución $i \in \varphi_\Pi(x)$, a ambas se le estipula un conjunto de soluciones vecinas $\mu_\Pi(i, x) \subseteq \varphi_\Pi(x)$. Hay que tener en cuenta que las vecinas no están especificadas explícitamente para cada solución i pero están determinadas indirectamente por x e i. Por lo cual, un problema de este tipo se basa en que: dado un caso x, se calcula una *solución óptima local*, esto es, una solución que no posea una vecina estrictamente mejor.

Dado un problema de optimización combinatoria OP produce distintos problemas de búsqueda local dependiendo de la función de entorno μ utilizada. Emplearemos la notación OP/μ para el correspondiente problema de búsqueda local. Las funciones de vecindad pueden ser bastante complicadas y no tienen porqué ser simétricas, es decir, i puede ser vecina de j pero no viceversa. Resulta claro que cuanto mayor sea el entorno a considerar, más difícil computacionalmente será tratarlo y, en consecuencia, de hallar su óptimo local (aunque será de mejor calidad). La situación idílica es conseguir encontrar el denominado entorno **exacto**, en el cual el problema

de búsqueda local coincide con el problema de óptimización más general pues en este se verifica que todo óptimo local es también óptimo global. Es esencial que seamos capaces de encontrar los entornos de modo eficiente, ya que no parece económico considerar un entorno demasiado extenso pues, como ya se ha comentado, provoca que sea mucho más difícil encontrar el óptimo local e incluso determinar los límites del vecindario.

Normalmente, se suelen asociar a la búsqueda local con problemas **NP-duros**, sin embargo, existen muchos problemas resolubles polinómicamente que pueden ser considerados como problemas de búsqueda local; esto es posible mediante la definición apropiada de una estructura exacta de entornos de modo que puedan ser encontrados de forma eficiente.

Lo dicho en cuanto a problemas de búsqueda local se reduce a probar un resultado de este tipo:

"Una solución no es óptima $\Leftrightarrow$ puede ser mejorada por un cierto tipo de perturbación".

Veamos entonces algunos ejemplos de problemas polinomiales de esta índole que puntualizan algunos de los principales problemas del estudio de la complejidad de la búsqueda local.

- **Programación lineal**. En este caso, la región factible es un politopo y si existe una solución óptima del problema habrá un vértice del politopo que será óptimo. De esta forma, la búsqueda local se traduce en que un vértice nunca resulta una solución óptima si, y solo si, existe un vértice adyacente con un mejor coste. Por lo cual, un vecindario exacto es el formado por los vértices adyacentes en el politopo. Algebraicamente, asumiendo que el problema lineal es no degenerado (recuérdese que son aquellos en el que dada una solución básica factible, el número de variables básicas es exactamente igual al número de restricciones activas, lo que asegura que cada conjunto de variables básicas se corresponde con un único vértice del poliedro factible), una solución factible no es óptima si puede ser mejorada cambiando una variable básica por una variable que no se encuentre en ella. Notemos que el algoritmo del *Simplex* es precisamente la búsqueda local que actúa de este modo tanto geométrica como algebraicamente.
- **Árbol de expansión mínima**. En este problema, la optimalidad no se cumple si, y solamente si, puede mejorarse añadiendo una arista al árbol y eliminando otra en el ciclo que se forma entonces. En consecuencia, un entorno exacto es aquel en el que dos árboles de expansión son vecinos si podemos obtener uno a partir del otro intercambiando una arista por otra (aunque basarse en esta caracterización para resolver este problema no es del todo eficiente).

Para rematar con esta sección, vamos a adentrarnos en la complejidad de los algoritmos heurísticos de búsqueda local a tratar mediante la definición de unos problemas asociados.

El problema local de optimalidad. Este problema consiste en averiguar si una solución dada para una cierta instancia de un problema es un óptimo local y en caso negativo se encuentra una solución vecina que la mejore.

Debido al modo en el que está definido este problema, está relacionado con la dificultad de ejecutar una iteración con nuestro heurístico. En los heurísticos de búsqueda local más comunes esto toma tiempo polinomial.

Para los dos problemas que presentaremos a continuación, es imprescindible la definición del algoritmo de búsqueda local "estándar". Tal algoritmo se encarga de, en presencia de una instancia x y empezando en una solución inicial i_0, moverse iterativamente desde una solución a otra vecina mejor, siempre que haya una; hasta que termina en la solución óptima local ι. En ciertas situaciones, el algoritmo de búsqueda local "estándar" debe decidir cual de los "mejores vecinos" escoge en caso de que posean los mismos costes. Una norma que permita seleccionar solamente un "mejor" vecino en cada iteración del heurístico considerado es una *regla pivotal*. En consecuencia, dado que algunas soluciones pueden tener más de un mejor vecino, una cierta función de vecindad no determina de manera única el algoritmo de búsqueda local "estándar" pues depende de la regla pivotal tomada. La elección de la regla pivotal puede afectar drásticamente a la complejidad del algoritmo de búsqueda local. Por lo tanto, procuraremos escoger algoritmos de búsqueda local con una regla pivotal cuya complejidad sea la mejor posible.

Problema del tiempo de ejecución: Determinar cual es el peor caso de complejidad temporal de un algoritmo de búsqueda local estándar con una regla pivotal dada.

Por último, nos interesa saber cuán rápidamente podemos encontrar la solución óptima local empleando un cierto algoritmo aplicable al problema tratado.

Problema del óptimo local estándar. Sea un caso x y una solución inicial i_0, encontrar el óptimo local producido por el algoritmo de búsqueda local estándar empezando desde i_0.

1.6. Teoría de grafos

Los grafos son las estructuras combinatorias fundamentales usadas a lo largo de este trabajo. En este capítulo, siguiendo [14], veremos las principales definiciones y notaciones.

1.6.1. Una revisión de conceptos básicos de grafos

Un **grafo** G **no dirigido** viene dado por un conjunto de **nodos** o **vértices** V y un conjunto E de pares no ordenados de vértices denominados **arcos** o **aristas** (en caso de ambigüedad sobre

el grafo al que se hace referencia, se denotarán como $V(G)$ o $E(G)$). Sabiendo esto, diremos que un arco une dos nodos. El grafo se dice **dirigido** o **digrafo** si los pares están ordenados. Cabe mencionar que V y E son conjuntos finitos.

Dos aristas que unen los mismos nodos se llaman **paralelas** y si un grafo no tiene aristas paralelas se llama **simple**. Dos nodos unidos por una arista se llaman **adyacentes** y decimos que los nodos son los **puntos finales** de la arista. También se dice que cada uno de los puntos finales de una arista **inciden** en ella.

Si un arco de un digrafo va de un nodo v a otro w, decimos que sale de v y entra en w. Además, si dos aristas comparten un nodo se llaman **adyacentes**.

Para lo que sigue, es importante la siguiente notación. Si v es un nodo de un grado dirigido, $\delta^+(v)$ es el conjunto de arcos que salen de v y $\delta^-(v)$ el conjunto de arcos que entran en v.

Por último, cabe destacar que un **subgrafo** de un grafo es otro grafo cuyos conjuntos de nodos y aristas están contenidos en los conjuntos de nodos y aristas, respectivamente, del grafo de partida.

Capítulo 2

El problema de flujo en redes

En este capítulo, tomando como referencia [14] y conceptos introducidos, consideraremos el problema de flujo en redes y emplearemos las notaciones definidas en los preliminares de teoría de grafos. Consideremos G un grafo dirigido con sus respectivos conjunto de nodos V y conjunto de aristas E. A estos, es necesario la adición de las llamadas *capacidades de las aristas* que vienen dadas por una aplicación $u : E \to \mathbb{R}^+$ que representa la cantidad máxima de unidades de flujo que se nos permite transportar a través de cada arista $e \in E$. Además, destacan por encima del resto dos nodos distinguidos: la **fuente** y el **sumidero**, que denotaremos como a y b, respectivamente. A la presencia de digrafo, capacidades de las aristas y nodos distinguidos se le llama en ocasiones **red**.

Con todo esto, desarrollaremos resultados que nos permitan estudiar algoritmos efectivos para lograr nuestro principal objetivo: conseguir el *flujo máximo*, es decir, descifrar cuál es el máximo número de unidades que podemos transportar a través de la red (G, u, a, b) (empezando en a y rematando en b). A continuación, aclaramos a qué nos referimos con el término "flujo" en estas circunstancias.

Definición 2.1. Consideremos una red (G, u, a, b). Se denomina **flujo** a una función $f : E \to \mathbb{R}^+$ que verifica la propiedad de que para cada arista e, $f(e) \leq u(e)$, es decir, el flujo enviado por cada arista no puede exceder su capacidad. Además, se dice que f satisface la **regla de conservación de flujo** en el vértice v si:

$$\sum_{e \in \delta^+(v)} f(e) = \sum_{e \in \delta^-(v)} f(e),$$

es decir, con la notación anterior se indica que el flujo que entra en el vértice v (a través de las aristas entrantes) es el mismo que el que sale (a través de las aristas salientes). Esto significa que ningún vértice se "queda" con nada, en otras palabras, los nodos centrales son sólo nodos de paso.

Dada una red, llamamos **a-b-flujo** a un flujo que satisface la propiedad de conservación de flujo en todos los vértices menos en la fuente y el sumidero. Para un tal flujo, f, se define su valor como sigue;

$$valor(f) := \sum_{e \in \delta^+(a)} f(e) - \sum_{e \in \delta^-(a)} f(e),$$

esto es, la diferencia entre el flujo que sale de la fuente y el que entra.

El problema que resulta el pilar fundamental del tratamiento de este capítulo se formula ahora.

PROBLEMA DEL FLUJO DE VALOR MÁXIMO	
Instancia:	Una red (G, u, a, b).
Objetivo:	Hallar un a-b-flujo máximo.

Las aplicaciones de este problema son inmensas tanto en ingeniería como logística. Por ejemplo, consideremos el *problema de asignación de trabajo*: sean n trabajos, los tiempos de duración $t_1, \ldots, t_n \in \mathbb{R}^+$ y un subconjunto no vacío $S_i \subseteq \{1, \ldots, m\}$ de empleados que pueden contribuir a cada trabajo $i \in \{1, \ldots, n\}$. Se definen entonces las variables $x_{ij} \in \mathbb{R}^+$, el tiempo empleado para cada trabajo i por cada trabajador $j \in S_i$. En cuanto a las restricciones, debe cumplirse que cada trabajo se realice al completo, lo que se traduce en que $\sum_{j \in S_i} x_{ij} = t_i$, $i = 1, \ldots, n$. Nuestro objetivo es minimizar la cantidad de tiempo en el cual los trabajos son hechos, esto es, $T(x) := \text{máx}_{j=1}^{m} \sum_{i:j \in S_i} x_{ij}$. Intuitivamente $T(x)$ refleja el tiempo que emplea el trabajador más sobrecargado. Se quiere minimizar este valor pues se busca que la distribución de la carga de trabajo no sea muy dispar y, sobre todo, que se realice lo antes posible. El problema así definido se puede resolver de modo sencillo tanto con *programación lineal* como mediante un algoritmo combinatorio y es esta segunda forma la que más nos interesa en nuestro caso.

Para encontrar el óptimo, $T(x)$, se puede aplicar la denominada búsqueda binaria de forma que para cada valor concreto, sea T, tendremos que encontrar valores y_{ij} reales positivos, para cada trabajo i cumpliendo que $\sum_{j \in S_i} y_{ij} = t_i$ y $\sum_{i:j \in S_i} y_{ij} \leq T$. Lo interesante es que podemos representar los conjuntos S_i por medio de un grafo dirigido bipartito con dos conjuntos de nodos, uno de los cuales representa los trabajos y otro los trabajadores. Para cada nodo representando un trabajo, asignamos una arista que lo conecta a todos los nodos representando trabajadores j, con $j \in S_i$. Para completar el grafo, se añade una fuente que se conecta a todos los nodos de los trabajos con capacidad el tiempo del correspondiente trabajo y por último un nodo sumidero conectado a los nodos que representan los trabajadores. La capacidad de estos últimos arcos, así como la de los que conectan trabajos a trabajadores, es T. Se cumple que cualquier solución factible del problema de asignación de trabajo, x, con $T(x) \leq T$ corresponde a un a-b-flujo de valor $\sum_{i=1}^{n} t_i$ en el grafo definido y se corresponde con el máximo flujo.

2.1. Teorema de máximo flujo mínimo corte

La definición del *problema del flujo máximo* sugiere la siguiente formulación LP (*linear program*).

$$\begin{aligned} \text{máx} \quad & \sum_{e\in\delta^+(a)} x_e - \sum_{e\in\delta^-(a)} x_e & \\ \text{s.a.} \quad & \sum_{e\in\delta^-(v)} x_e = \sum_{e\in\delta^+(v)} x_e & \text{para todo } v \in V \setminus \{a,b\} \\ & x_e \leq u(e) & \text{para todo } e \in E \\ & x_e \geq 0 & \text{para todo } e \in E \end{aligned}$$

Dado que este problema de programación lineal está obviamente acotado (las capacidades y la cantidad de flujo son siempre finitos) y el flujo cero $f \equiv 0$ es siempre factible, siempre tiene óptimo.

Además, existe un algoritmo en tiempo polinomial que lo resuelve. Sin embargo, no estamos buscando esto sino que queremos un algortimo combinatorio. Ahora se introducirá una definición necesaria para lo que sigue.

Definición 2.2. Dada una red (G, u, a, b) y un par $(A, B) \subset V \times V$ que verifica las siguientes propiedades.

- La unión de los conjuntos del par da lugar al conjunto total de vértices V,
- la intersección de dichos conjuntos es vacía,
- la fuente de la red es un elemento de A y
- el sumidero de la red es un elemento de B.

Entonces el conjunto de aristas del conjunto $\{\{x, y\} \in E : x \in A\backslash B, y \in B\backslash A\}$ se llama ***a-b*-corte**.

Por tanto, dado un subconjunto de nodos de la red que contiene a la fuente, un a-b-corte no es más que un conjunto de aristas que conectan nodos de ese conjunto con nodos del conjunto de nodos complementario (el cual debe de contener al nodo sumidero), siendo denominada la suma de las capacidades de esas aristas la **capacidad** del corte. El problema que nos planteamos ahora es encontrar el a-b-corte de capacidad mínima.

Lema 2.3. *Consideremos una red y un subconjunto del conjunto de nodos, A, que contenga a la fuente, a, pero no al sumidero, t. Sea ahora un a-b-flujo, f. Se cumple lo siguiente respecto del valor de f.*

(a) *valor*$(f) = \sum_{e\in\delta^+(A)} f(e) - \sum_{e\in\delta^-(A)} f(e)$.

(b) *valor*$(f) \leq \sum_{e\in\delta^+(A)} u(e)$.

Demostración. Para demostrar la parte (a) basta tener en cuenta que en el nodo fuente se tiene que la diferencia entre el flujo que sale y el que entra es el valor del flujo f. Si ahora tenemos en cuenta que en los nodos restantes se cumple la propiedad de conservación de flujo, se sigue el resultado.

A partir de aquí, como el flujo que circula por cada arista es positivo y no excede la capacidad de la arista, se tiene la parte (b) del lema. □

En otras palabras, el valor de máximo flujo no puede superar la capacidad del a-b-corte de capacidad mínima. Para ver que además tenemos la igualdad, usamos el siguiente concepto.

Definición 2.4. Dado un grafo dirigido G, un **a-b-camino** es una secuencia de aristas que empieza en el vértice a y termina en el vértice b.

Definición 2.5. Dado un grafo dirigido G y una arista cualquiera de dicho grafo $e = (v, w)$ que va de v a w, definimos $\overleftarrow{e}$ como una nueva arista de w a v que recibe el nombre de **arista inversa** de e. Cabe destacar que una arista y su inversa son aristas paralelas.

Consideremos ahora una red, dada por un grafo dirigido G y la función u que define las capacidades de los arcos. Si ahora f es un flujo que recorre la red, se definen las **capacidades residuales** mediante una función u_f dada de la siguiente forma para cada arista e de G:

$$u_f(e) := u(e) - f(e) \quad \text{y} \quad u_f(\overleftarrow{e}) := f(e).$$

Es decir, para las aristas de partida, la capacidad residual es el margen que queda para el paso de más flujo y en las aristas inversas, representa el flujo enviado que se puede cancelar.

El grafo formado por los nodos de G y el conjunto de sus aristas y sus aristas inversas con capacidad residual estrictamente positiva, se denomina **grafo residual** y se denota G_f.

Dado un flujo f y un camino (o circuito) P en G_f, **aumentar** f a lo largo de P en γ (γ será un valor que describiremos a continuación) significa hacer lo siguiente para cada arista e perteneciente al conjunto aristas de P.

- En el caso de una arista del grafo, se incrementa el flujo en dicha arista en la cantidad γ.

- En el caso de una arista cuya inversa pertenezca al grafo residual, se disminuye el flujo de la arista en la cantidad γ.

Dada una red (G, u, a, b) y un a-b-flujo f, un **camino de aumento** f es un a-b-camino en el grafo residual G_f.

Teniendo en cuenta todo lo anterior, estamos preparados para definir el algoritmo de *Ford-Fulkerson* que trata de modo efectivo el problema del flujo máximo.

Algoritmo de Ford-Fulkerson

Entrada: Una red (G, u, a, b).
Salida: Un a-b-flujo f máximo.

① Asignar a cada arista del grafo un flujo igual a cero.

② Buscar P, un camino de aumento f. Si no existe, **detenerse**.

③ Tomar el valor γ como el valor mínimo de las capacidades residuales de todas las aristas. Aumentar f a lo largo de P en γ, y volver al paso ②.

A las aristas en las cuales se alcanza el mínimo de ③ se les llama a veces aristas de cuello de botella. El hecho de elegir así γ garantiza que f continúe siendo un flujo dado que así siempre se respetaran las capacidades de las aristas. Por definición de camino de aumento, es obvio que P es un a-b-camino y por lo tanto salvo en los nodos distinguidos a y b, en el resto se cumple la regla de conservación de flujo.

Encontrar un camino de aumento es fácil dado que solo se debe encontrar cualquier a-b-camino de G_f.

Con el algoritmo de *Ford-Fulkerson* definido de este modo, tomando ciertas capacidades irracionales problemáticas, el algoritmo puede no finalizar nunca.

Un ejemplo de un caso problemático se pretende ilustrar en la Figura 2.1, donde incluso tomando las capacidades de arista como valores naturales $N \in \mathbb{N}$, tenemos un número exponencial de aumentos. Podemos aumentar el flujo en una sola unidad en cada iteración si escogemos un camino de aumento de longitud exactamente 3 en cada una de ellas, por lo que necesitamos $2N$ iteraciones. Obsérvese que la longitud de la entrada es $O(\log N)$, dado que las capacidades están codificadas en forma binaria. Por lo que, en el peor caso, el algoritmo no tiene complejidad polinomial.

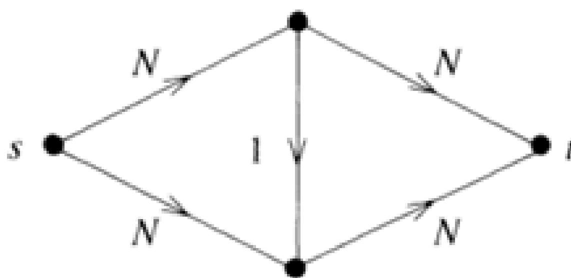

Figura 2.1: ejemplo de red.

Es imprescindible cerciorarse de que en el caso no problemático, en el que el algoritmo termina sin problemas, entonces el flujo resultante f es un flujo máximo. Con este objetivo, se presenta el siguiente resultado.

Teorema 2.6. *La propiedad de maximalidad de un a-b-flujo f es equivalente a la no existencia de un camino de aumento de f.*

Demostración. Veamos en primer lugar la implicación hacia la derecha. Para ello, emplearemos el contrarrecíproco. Supongamos que existe un camino de aumento P en el grafo residual G_f. El paso ③ del algoritmo de *Ford-Fulkerson* incrementa el flujo actual, produciendo un flujo con valor mayor. Por lo tanto, el flujo no puede ser máximo.

Ahora comprobaremos la otra implicación. En el caso en el que no exista ningún camino de aumento, entonces no hay forma de llegar desde el nodo fuente a hasta el nodo sumidero b. Definimos A como el conjunto de vértices que son alcanzables desde a. Por cómo está construido G_f, esto se traduce en que para cada arista que sale de algún nodo de A en el grafo original, el flujo ha alcanzado su capacidad máxima. También, a través de cada arista que entra en algún vértice de A, no se está enviando nada de flujo. Ahora, de acuerdo con la parte (a) del Lema 2.3, esto implica que la suma de las capacidades de todas las aristas que salen de A coincide con el $valor(f)$. Por lo tanto, aplicando la parte (b) del mismo lema, se concluye que este flujo f es de valor máximo. □

Corolario 2.7. *Sea un a-b-flujo de valor máximo, f, entonces existe un a-b-corte cuya capacidad es igual al $valor(f)$.*

Demostración. Se obtiene de modo inmediato teniendo en cuenta el Teorema 2.6 y la definición de *a*-*b*-corte. □

Junto con el Lema 2.3 (b), este hecho produce el resultado central de la teoría de flujo en redes, el Teorema del Flujo Máximo-Corte Mínimo (si se quiere ver la demostración completa, consúltese [9]).

Teorema 2.8 (Teorema del Flujo Entero)**.** *Dada una red, esto es, un grafo con capacidades en sus aristas, si estas capacidades son valores enteros, entonces podemos encontrar un flujo máximo cuyo valor también es entero.*

Demostración. Si se cumple la condición de que las capacidades de una red son enteras, es evidente que la cantidad γ determinada en el tercer paso del algoritmo de *Ford-Fulkerson* siempre es un valor entero por las propiedades del mínimo. Deducimos entonces que el algoritmo concluye en un número finito de pasos, pues sabemos que siempre existe un flujo máximo de valor finito y se cumple la afirmación del teorema. □

Cabe destacar que el resultado previo puede demostrarse de modo alternativo utilizando la unimodularidad total de la matriz de incidencia de un grafo dirigido, recordando que una matriz es totalmente unimodular si el determinante de cualquier submatriz cuadrada de ella es 0, +1 o -1.

Cerramos este estudio del problema de flujo en redes y para ello, presentaremos un resultado que resulta muy útil en la práctica, pues nos permite descomponer cualquier flujo de un modo muy característico. Este hecho, tiene grandes aplicaciones en problemas reales de optimización.

Teorema 2.9 (Teorema de Descomposición de Flujo)**.** *Dada una red y un a-b-flujo, f, de ella. Entonces, este flujo se puede descomponer a partir de una combinación de caminos y circuitos de manera que existe un conjunto X de caminos de a a b y un conjunto Y de circuitos junto con una función de pesos $w : X \cup Y \to \mathbb{R}^+$, tal que para cada arista $e \in E$, el flujo total $f(e)$ puede expresarse como la suma de los pesos de los caminos y circuitos que la contienen. Además, la suma de los pesos de todos los caminos de X es igual al valor(f) y el número total de caminos y circuitos necesarios en esta descomposición es menor o igual que el número de aristas del grafo.*

Cabe destacar que si el flujo f es entero, entonces también se puede escoger la función de pesos, w, con valores enteros.

Demostración. Vamos a construir los conjuntos X, Y y la función de pesos w por inducción sobre la cantidad de aristas del grafo que tienen un flujo distinto de cero.

Tomemos una arista $e = (v_0, w_0)$ tal que $f(e) > 0$. A partir de aquí, seguimos una trayectoria hacia delante mientras que el flujo lo permita, es decir, si $w_0 \neq b$, entonces debe existir una arista saliente de w_0, que denotaremos por (w_0, w_1), con flujo positivo. Consideremos $i = 1$ y continuamos el siguiente proceso iterativamente: en cada paso, si el vértice siguiente w_i ya ha aparecido en la secuencia $\{b, v_0, w_0, \ldots, w_{i-1}\}$ o si llegamos a b, paramos. Si no, buscamos una arista saliente de w_i con flujo positivo, avanzamos al siguiente vértice y repetimos el proceso. Como hay un número finito de vértices, el procedimiento debe finalizar en como máximo n pasos.

Realizamos un proceso análogo en la dirección contraria: si $v_0 \neq a$, buscamos una arista entrante (v_1, v_0) con flujo positivo y seguimos hacia atrás, de forma sucesiva, hasta llegar a a o encontrar un ciclo.

Al final de este proceso, habremos construido o bien un camino de a hasta b, o bien un circuito, usando solamente aristas con flujo positivo. Denotamos esto como A. Definimos el peso $w(A)$ como el mínimo flujo sobre las aristas de A. A continuación, definimos un nuevo flujo f' restando $w(A)$ al flujo de todas las aristas que están en A y dejando el resto igual. Este nuevo flujo da lugar a menos aristas con flujo no nulo que el flujo original, así que por hipótesis de inducción, podemos aplicar el mismo procedimiento a f'. Al sumar la trayectoria A con peso $w(A)$ a la descomposición obtenida por inducción para f', obtenemos la construcción que buscábamos. $\square$

Capítulo 3

El problema de la mochila

En este capítulo nos encargaremos de estudiar, teniendo en cuenta [14], el siguiente problema que resulta ser, de algún modo, el problema NP-completo más "fácil".

Se consideran n objetos disponibles con valores o ganancias no negativas asociadas $c_1, \ldots, c_n$ y pesos no negativos $w_1, \ldots, w_n$. Se dispone de una mochila que puede soportar un peso máximo de W. El **problema de la mochila** consiste en encontrar un subconjunto de los n elementos cuya suma de pesos no exceda la capacidad de la mochila y haga máxima la suma de las ganancias

La resolución de este problema tiene grandes aplicaciones cuando nos encontramos en situaciones en las cuales queremos seleccionar un subconjunto óptimo de peso acotado sobre un conjunto de elementos que tienen un peso y una ganancia.

3.1. Algunas modificaciones del problema de la mochila

Una modificación del problema de la mochila, partiendo de los mismo elementos que en éste, consiste en encontrar números $x_1, \ldots, x_n \in [0, 1]$ tal que $\sum_{j=1}^{n} x_j w_j \leq W$ y $\sum_{j=1}^{n} x_j c_j$ es máximo. Esta modificación se denomina **problema fraccionado de la mochila**.

Se diferencia con el problema de la mochila en que en este caso se pueden coger fracciones de los objetos. Mientras que el problema anterior era **NP-completo**, este se puede resolver en tiempo polinomial como veremos.

Un algoritmo que resuelve este problema es consecuencia de la Proposición 3.1.

Proposición 3.1. *Sean $c_1, \ldots, c_n, w_1, \ldots, w_n$ y W enteros no negativos (los valores de **eficiencia** de cada objeto) con*

$$\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n},$$

y sea

$$k := \min\{j \in \{1,\dots,n\} : \sum_{i=1}^{j} w_i > W\}.$$

Entonces una solución óptima de la instancia dada del problema de la mochila fraccionada se define como

$$\begin{aligned} x_j &:= 1 && \textit{para } j = 1,\dots,k-1, \\ x_k &:= \frac{W - \sum_{j=1}^{k-1} w_j}{w_k}, \\ x_j &:= 0 && \textit{para } j = k+1,\dots,n. \end{aligned}$$

La solución se basa en ir llenando la mochila por orden de eficiencia con objetos enteros. En el momento que tomando este orden se exceda el valor W, pues se toma la parte correspondiente de este último para que el total sea W, es decir, se coge $(W - \sum_{j=1}^{k-1} w_j)/w_k$. En [8] se pueden encontrar más detalles.

Ordenar los elementos toma un tiempo $O(nlogn)$ (los principales algoritmos de ordenación tienen esta complejidad) y computar k puede hacerse en tiempo $O(n)$ mediante simple exploración lineal. No obstante, este algoritmo no es el mejor en cuanto a eficiencia computacional. A continuación, se presenta un concepto que ofrece otra interpretación del problema.

Definición 3.2. Tomemos un problema de la mochila como se definió al principio del capítulo y n números reales $z_1,\dots,z_n$. Entonces, la **mediana ponderada** por $(w_1,\dots,w_n;W)$ con respecto a $(z_1,\dots,z_n)$ se define como el único número $z^* \in \{z_1,\dots,z_n\}$ para el cual $\sum_{i:z_i<z^*} w_i < W \leq \sum_{i:z_i\leq z^*} w_i$.

Es decir, la mediana ponderada se corresponde con el valor mínimo tal que al sumar los pesos hasta z^* (incluyéndolo), se alcanza al menos W.

Encontrar esta mediana ponderada constituye el **problema de la mediana ponderada**.

El problema de selección se puede ver como un caso particular del problema de la mediana ponderada.

PROBLEMA DE SELECCIÓN	
Instancia:	Un entero n, números $z_1,\dots,z_n \in \mathbb{R}$, y un entero $k \in \{1,\dots,n\}$.
Objetivo:	Encontrar el k-ésimo número más pequeño entre $z_1,\dots,z_n$.

Efectivamente, el **problema de selección** es un caso particular del problema de la mediana ponderada, pues basta tomar en este último $w_i := 1$ para $i = 1,\dots,n$ y $W = k$.

La mediana ponderada se puede determinar en tiempo $O(n)$ y a continuación mostramos un algoritmo que lo permite.

ALGORITMO DE LA MEDIANA PONDERADA

Entrada: Un entero n, números $z_1, \ldots, z_n \in \mathbb{R}$, $w_1, \ldots, w_n \in \mathbb{R}^+$ y un umbral W con $0 < W \leq \sum_{i=1}^{n} w_i$.
Salida: La mediana ponderada por $(w_1, \ldots, w_n; W)$ con respecto a $(z_1, \ldots, z_n)$.

① Dividir el conjunto de los n números z_i en bloques contiguos de hasta cinco elementos cada uno. Ordenar los elementos de cada grupo y tomar la mediana de cada bloque. Esto proporciona una nueva lista M de $\lceil \frac{n}{5} \rceil$ elementos medianos.

② Aplicar recursivamente el mismo procedimiento de ① sobre la lista M para hallar su mediana z_m.

③ Reordenar convenientemente los elementos y suponer que $z_i < z_m$ si $i \leq k$, $z_i = z_m$ si $k + 1 \leq i \leq l$ y $z_i > z_m$ si $i \geq l + 1$.

④ Si W, la capacidad de la mochila, es mayor que la suma de los k primeros pesos w_i pero menor o igual que la suma de los l primeros, entonces la mediana ponderada es $z^* := z_m$. **Detenerse**.

Si se cumple que la suma de los l primeros pesos es estrictamente menor que la capacidad de la mochila, entonces ajustar el umbral y restringir la búsqueda a los elementos superiores. Esto es, aplicar el proceso recursivo de la mediana ponderada por $\left(w_{l+1}, \ldots, w_n; W - \sum_{i=1}^{l} w_i\right)$ con respecto a $(z_{l+1}, \ldots, z_n)$. **Detenerse**.
Si se tiene que

$$\sum_{i=1}^{k} w_i \geq W$$

entonces mantener el umbral y restringir la búsqueda a los elementos menores. Para ello, aplicar el proceso recursivo de la mediana ponderada por $(w_1, \ldots, w_k; W)$ con respecto a $(z_1, \ldots, z_k)$. **Detenerse**.

Teorema 3.3. *El **método anterior** es correcto y puede completarse en un tiempo proporcional al tamaño de la entrada, esto es, en tiempo lineal $O(n)$.*

Demostración. La corrección del algoritmo es fácil de verificar. Para verlo, es necesario que el algoritmo cumpla con dos cosas clave: que el algoritmo siempre termine y de una salida válida, y que la salida cumpla la definición de mediana ponderada.

Veamos que dado $z_1, \ldots, z_n \in \mathbb{R}$, $w_1, \ldots, w_n \in \mathbb{R}^+$ y $W \in \mathbb{R}$ con $0 < W \leq \sum_{i=1}^{n} w_i$, el

algoritmo de la mediana ponderada encuentra correctamente el número $z^* \in \{z_1, \ldots, z_n\}$ tal que:

$$\sum_{i:z_i<z^*} w_i < W \leq \sum_{i:z_i \leq z^*} w_i.$$

El algoritmo selecciona un *pivote* z_m calculado como la mediana de un conjunto de medianas de bloques de tamaño 5. Esta elección garantiza que al menos una fracción constante del conjunto es menor o mayor que z_m, asegurando que haya progreso en cada llamada recursiva.

El conjunto de valores $z_1, \ldots, z_n$ se divide en tres grupos:

$$A := \{i : z_i < z_m\},$$
$$B := \{i : z_i = z_m\},$$
$$C := \{i : z_i > z_m\}.$$

Se calculan los pesos acumulados correspondientes:

$$W_A := \sum_{i \in A} w_i, \quad W_B := \sum_{i \in B} w_i, \quad W_C := \sum_{i \in C} w_i.$$

Se comparan W con W_A y $W_A + W_B$:

- Si $W_A < W \leq W_A + W_B$, entonces $z^* = z_m$ cumple:

  $$\sum_{i:z_i<z_m} w_i < W \leq \sum_{i:z_i \leq z_m} w_i,$$

  que es exactamente la definición de mediana ponderada. El algoritmo se detiene correctamente.

- Si $W > W_A + W_B$, entonces la solución debe encontrarse entre los $z_i > z_m$. Definimos un nuevo subproblema sobre los elementos de C, con nuevo umbral $W' := W-(W_A+W_B)$. Este subproblema conserva la definición, pues se ha eliminado peso insuficiente para alcanzar W.

- Si $W \leq W_A$, entonces la solución está entre los $z_i < z_m$. Se resuelve recursivamente el problema sobre los elementos de A, con el mismo umbral W.

En cada llamada recursiva, el tamaño del problema disminuye en una fracción constante del total, ya que el algoritmo de selección mediana de medianas garantiza que al menos 30 % de los elementos son eliminados.

Ahora veamos el tiempo de ejecución. Emplearemos como notación $t(m)$ para el peor caso de tiempo de ejecución para m elementos. Con todo, se deduce:

$$t(m) = O(m) + t\left(\left\lceil \frac{m}{5} \right\rceil\right) + O(m) + t\left(\frac{1}{2}\left\lceil \frac{m}{5} \right\rceil 5 + \frac{1}{2}\left\lceil \frac{m}{5} \right\rceil 2\right),$$

porque la llamada recursiva en el paso ④ descarta al menos tres elementos de al menos la mitad de los bloques de cinco elementos. La fórmula de recurrencia anterior implica que $t(m) = O(m)$: dado que $\lceil \frac{m}{5} \rceil \leq \frac{9}{41}m$ para todo $m \geq 37$, se obtiene $t(m) \leq cm + t\left(\frac{9}{41}m\right) + t\left(\frac{7}{2}\frac{9}{41}m\right)$ para una constante c adecuada y $m \geq 37$. Con esto, se puede verificar fácilmente por inducción que $t(m) \leq (82c + t(36))m$. Por lo tanto, el tiempo de ejecución total es lineal.

□

Así, se obtiene un resultado para el tiempo de los problemas de selección y de la mochila fraccional.

Corolario 3.4. *Se verifican las siguientes propiedades.*

a) El problema de selección puede resolverse en un tiempo proporcional al tamaño de la entrada, es decir, en tiempo lineal $O(n)$.

b) El problema fraccionado de la mochila puede resolverse en un tiempo proporcional al tamaño de la entrada, esto es, en tiempo lineal $O(n)$.

Demostración. El apartado (a) se prueba fácilmente aplicando el Teorema 3.3 y tomando como peso para cada objeto el valor 1 y un valor de k entero para la capacidad de la mochila.

La parte (b) se concluye teniendo en cuenta, como ya advertimos al inicio de esta sección (tras la Proposición 3.1), que ajustar $z_i := \frac{c_i}{w_i}$ $(i = 1, \ldots, n)$ reduce el problema de la mochila fraccional al problema de la mediana ponderada. □

3.2. Resolución del problema de la mochila

En lo que queda de capítulo, se estudia la resolución del problema de la mochila inicial. Las técnicas de las anteriores secciones son usadas de algún modo aquí. Además, cabe recordar que un *algoritmo de factor n* es aquel cuya solución nunca será peor que n veces el valor de la solución óptima del problema a tratar.

Proposición 3.5. *Consideremos el problema de la mochila tal como se definió al inicio del capítulo, con las restricciones adicionales de que todos los parámetros (beneficios, pesos y capacidad de la mochila) son enteros, los pesos son menores o iguales que la capacidad de la mochila y*

además

$$\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n}.$$

Se define, también, un entero no negativo k como el menor valor tal que al considerar los k primeros objetos, el peso acumulado supera la capacidad de la mochila.

Entonces, un algoritmo de aproximación de factor 2 para el problema se reduce a elegir lo mejor entre seleccionar los $k-1$ primeros objetos o bien el k-ésimo (nótese que ambas posibilidades son factibles). Además, este algoritmo se puede ejecutar en un tiempo lineal O(n).

Demostración. Tomemos un problema de la mochila arbitrario. Cualquier artículo i con $w_i > W$ no es útil y puede eliminarse de antemano. El número k puede calcularse en tiempo $O(n)$ sin ordenar: esto es justamente un problema de la mediana ponderada como se describió anteriormente (Teorema 3.3).

Por la Proposición 3.1, $\sum_{i=1}^{k} c_i$ es una cota superior para el valor óptimo del problema de la mochila fraccional y ,por lo tanto, también para el problema de la mochila entero. Por tanto, lo mejor entre seleccionar los $k-1$ primeros artículos o bien el artículo k, da lugar a un valor del objetivo (maximizar las ganancias correspondientes a los artículos seleccionados) que no es inferior al 50 % del óptimo. □

Previamente a la introducción de un algoritmo exacto, se proporciona un resultado relativo a su complejidad.

Teorema 3.6. *Las soluciones del problema de la mochila pueden verificarse en tiempo polinomial.*

Demostración. Vamos a demostrar que el problema de decisión relacionado, definido como sigue, es de **P**. Dados enteros no negativos n, $c_1, \ldots, c_n$, $w_1, \ldots, w_n$, W y K, y $S \subseteq \{1, \ldots, n\}$ una solución candidata, ¿S verifica que $\sum_{j \in S} w_j \leq W$ y $\sum_{j \in S} c_j \geq K$?

Este problema de decisión claramente pertenece a **P** ya que una solución candidata (el subconjunto S) puede representarse como una secuencia binaria $(x_1, \ldots, x_n) \in \{0,1\}^n$ donde $x_j = 1$ si $j \in S$ y 0 en caso contrario. Dada una solución candidata, se pueden verificar las dos condiciones

$$\sum_{j=1}^{n} x_j w_j \leq W \quad \text{y} \quad \sum_{j=1}^{n} x_j c_j \geq K$$

en tiempo polinomial en n, lo cual implica que el problema de decisión es de P. En consecuencia, nuestro problema original de la mochila pertenece a NP. □

Ahora, presentaremos un teorema más fuerte pero sin demostración.

Teorema 3.7. *El problema de la mochila es **NP**-duro.*

Dado que no hemos demostrado que el problema de la mochila sea fuertemente **NP**-duro (un problema es fuertemente **NP**-duro si sigue siendo **NP**-duro incluso cuando todos sus números de entrada están acotados por una función polinomial del tamaño de la entrada), existe la posibilidad de un algoritmo pseudopolinomial.

Definición 3.8. Dado P un problema de decisión o un problema de optimización tal que cada instancia consiste en un lista de números enteros y denotando como *mayor(x)* al mayor de estos enteros, un algoritmo para P se denomina **pseudopolinómico** si su tiempo de ejecución está acotado por un polinomio en *tamaño(x)* y *mayor(x)*.

A continuación, mostraremos cómo obtener un algoritmo con tiempo de ejecución $O(nC)$, donde $C := \sum_{j=1}^{n} c_j$. Describimos este algoritmo de manera directa, sin construir un grafo ni referirnos a caminos más cortos. Dado que la corrección del algoritmo se basa en fórmulas recursivas simples, hablamos de un algoritmo de programación dinámica. Se debe básicamente a [3] y a [8].

Explicamos el algoritmo de programación dinámica para el problema de la mochila de manera intuitiva. Se parte de un problema de la mochila con n artículos y enteros W, w_i y c_i representando capacidad, pesos y ganancias. La salida será el subconjunto S de artículos que maximiza las ganancias y no excede la capacidad permitida. Se toma C como la suma de todas las ganancias, y se consideran las variables $x(j,k)$, que representan el peso mínimo del subconjunto S de los j primeros elementos cuya ganancia es k. La clave está en el cálculo iterativo de estas variables:

$$x(j,k) = \begin{cases} x(j-1,k-c_j)+w_j & \text{si } c_j \leq k \text{ y } x(j-1,k-c_j)+w_j \leq \text{mín}\{W,\, x(j-1,k)\} \\ x(j-1,k) & \text{en otro caso} \end{cases}$$

para j desde 1 hasta n y k desde 0 hasta C, con $x(0,0)=0$ y $x(0,k)=\infty$ para $k=1,\ldots,C$. Por medio de variables auxiliares $s(j,k)$, se guarda cuál de los dos casos se cumple en la definición de $x(j,k)$. El algoritmo va almacenando los distintos subconjuntos, salvo que no sean factibles o exista otro subconjunto de artículos con la misma ganancia pero menos peso. Por último, se elige, de entre todos los subconjuntos, aquel con ganancia máxima que no supere la capacidad de la mochila.

Se tiene, como consecuencia, el siguiente resultado.

Teorema 3.9. *El* ***algoritmo de programación dinámica*** *para el problema de la mochila encuentra una solución óptima en tiempo $O(nC)$.*

Demostración. Veamos únicamente lo referido al tiempo de ejecución. Para comenzar asignamos, como se ha explicado, $x(0,0) := 0$ y $x(0,k) := \infty$ para $k=1,\ldots,C$. Esto nos lleva un tiempo de ejecución de $O(C)$.

Para cada objeto $j = 1, \ldots, n$, se recorren todos los valores posibles de ganancia $k = 1, \ldots, C$. Para cada par (j, k), se realiza un número constante de operaciones (asignaciones, sumas, comparaciones); por lo tanto, esta etapa requiere un tiempo de $O(nC)$.

Se recorre hacia atrás el conjunto de subconjuntos para recuperar el subconjunto óptimo, lo cual toma a lo sumo n pasos, es decir, tiempo $O(n)$.

Sumando las tres fases del algoritmo, el tiempo total es:

$$O(C) + O(nC) + O(n) = O(nC)$$

ya que es evidente que $O(C) \subseteq O(nC)$ y $O(n) \subseteq O(nC)$.

Por lo tanto, el algoritmo tiene tiempo de ejecución $O(nC)$. $\square$

Con el objetivo de que el algoritmo que acabamos de definir sea más eficiente, podemos ejecutar previamente el algoritmo de aproximación de factor 2 de la Proposición 3.5 pues, al multiplicar por dos su solución, podemos obtener una cota superior C más ajustada que $\sum_{i=1}^{n} c_i$.

La cota $O(nC)$ no es polinomial en el tamaño de la entrada, ya que dicha entrada solo puede acotarse por $O(n \log C + n \log W)$, asumiendo que $w_j \leq W$ para todo j. No obstante, se dispone de un algoritmo pseudopolinomial que puede resultar bastante eficiente siempre que los valores numéricos involucrados no sean demasiado grandes. En particular, si tanto los pesos $w_1, \ldots, w_n$ como los beneficios $c_1, \ldots, c_n$ son pequeños, el algoritmo de complejidad $O(nc_{\text{máx}} w_{\text{máx}})$ propuesto en [18] es el más rápido, donde se tiene que $c_{\text{máx}} = \text{máx}\{c_1, \ldots, c_n\}$ y $w_{\text{máx}} = \text{máx}\{w_1, \ldots, w_n\}$.

Capítulo 4

El problema del viajante de comercio

El problema del viajante de comercio (TSP por sus siglas en inglés) ha sido estudiado por ejemplo en [1]. Se caracteriza por un conjunto de ciudades y la distancia entre cada par de ellas. El objetivo es diseñar un tour que, empezando en una ciudad, las recorra todas y vuelva a la ciudad de partida. La solución del problema es el tour que recorra la distancia mínima.

Los algoritmos de búsqueda local, como los considerados en capítulos anteriores, son también aquí una herramienta útil siendo preciso adaptar al nuevo contexto la estructura de vecindad considerada y las modificaciones admisibles. En [14] se menciona que puede dar buenos resultados iniciar la búsqueda local con una solución inicial creada por otro heurístico.

En la construcción de algoritmos puede ser conveniente representar el problema con un grafo, donde los nodos son las ciudades y las aristas son conexiones entre pares de ciudades.

Un algoritmo muy popular es el denominado ***k*-opt** ($k \geq 2$), que se explica a continuación.

Se parte de un grafo completo que representa las ciudades y todas las posibles conexiones, además de un coste (por ejemplo, una distancia) asociado a cada arista. El algoritmo sigue estos pasos:

① Partir de un tour inicial, elegido aleatoriamente o por algún heurístico, sea P.

② Considerar la familia, $\mathcal{S}$, de todos los subconjuntos de k aristas de P.

③ Para todo $S \in \mathcal{S}$, considerar todos los tours, P', que incluyen todas las aristas de P excepto las pertenecientes a S. Siempre que el coste de P' sea menor que el de P, hacer $P = P'$ y volver a ②.

④ Parar y proporcionar el último tour P.

Se denomina **tour k óptimo** aquel que no puede mejorar el algoritmo *k-opt*. La Figura 4.1

extraída de [14], muestra distancias euclídeas y el caso de un tour 2 óptimo pero no 3 óptimo.

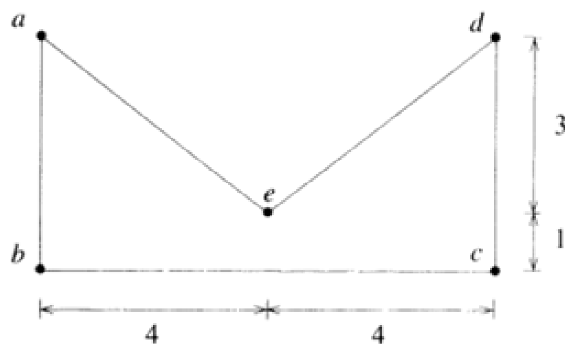

Figura 4.1: ejemplo de tour.

Nótese que (a, b, e, c, d, a) es el óptimo.

En el ejemplo mostrado en el lado derecho de la Figura 4.2 tenemos un tour 3 óptimo pero no

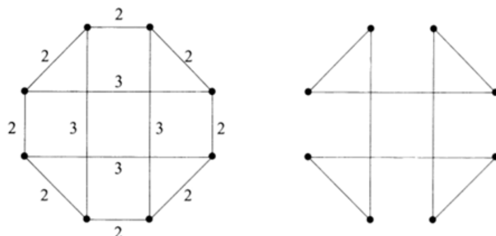

Figura 4.2: ejemplo de tour.

4 óptimo. En el lado izquierdo, se muestran los costes de algunas aristas, siendo 4 el coste de las que no se muestran. Se puede ver que en este caso, donde las distancias cumplen la desigualdad triangular, el algoritmo *4-opt* proporciona la solución óptima.

Un problema que se plantea en el algoritmo *k-opt* es la elección de la k, por lo que en [15] se diseña un algoritmo que lo determina sobre la marcha del propio algoritmo. Antes de presentarlo, se proporcionan varios conceptos y un resultado.

Definición 4.1. Consideremos un grafo completo con n nodos (ciudades), una función c que define el coste de cada arista y un tour T. Un **camino alternante** es un camino que pasa por las ciudades $x_0, x_1, \ldots, x_{2m}$ de forma que no se repite ninguna arista y la arista (x_i, x_{i+1}) es una arista de T si, y solo si, i es par, para $i = 0, \ldots, 2m-1$. El camino alternante es **cerrado** si la primera y la última ciudad visitadas son la misma.

Si denotamos por P a un camino alternante, la **ganancia** de P se define como

$$g(P) = \sum_{i=0}^{m-1} c(x_{2i}, x_{2i+1}) - c(x_{2i+1}, x_{2i+2}).$$

P de denomina **propio** si $g((x_0, \ldots, x_{2l})) > 0$ para todo $l = 1, \ldots, m$. Al conjunto de aristas de P, lo denotaremos por $E(P)$.

Ejemplo 4.2. Tomando el tour que aparece en la Figura 4.1, tenemos que (a, e, b, c, e, d, a) es un camino alternante, cerrado y propio que nos permite ver que un mismo vértice puede ser visitado más de una vez en un camino alternante.

El siguiente resultado fue probado por Lin y Kernighan.

Lema 4.3. *Consideremos un grafo completo con n ciudades, la función de coste de las aristas c y un tour T. Supongamos que P es un camino alternante cerrado con ganancia $g(P)$, estrictamente positiva, y supongamos que el conjunto de nodos de T con aristas la diferencia simétrica de los conjuntos de aristas de T y P, $(V(T), E(T)\triangle E(P))$, es también un tour denominado T'. Entonces, $c(T') = c(V(T), E(T)\triangle E(P)) < c(T)$, es decir, el coste del nuevo tour es menor que el del tour de partida. Además, existe un camino alternante cerrado propio , Q, cuyo conjunto de aristas es el mismo que el de P.*

Demostración. Para probar la primera parte, sean T un tour y $P = (x_0, x_1, \ldots, x_{2m} = x_0)$ un camino alternante cerrado como en el enunciado del lema. Definimos $A := \{\{x_{2l}, x_{2l+1}\} : l = 0, \ldots, m-1\} \subseteq E(T)$, como las aristas de P que están en T y $B := \{\{x_{2l+1}, x_{2l+2}\} : l = 0, \ldots, m-1\} \subseteq E(P) \setminus E(T)$, como las aristas de P que no están en T. Por tanto, por la definición de la ganancia de P, $g(P)$, ésta será igual a la siguiente expresión

$$\sum_{e \in A} c(e) - \sum_{e \in B} c(e).$$

Si ahora consideramos el tour T como en el enunciado del lema, su conjunto de aristas es justamente $(E(T) \setminus A) \cup B$.

Entonces, el coste del nuevo tour es:

$$c(T') = \sum_{e \in E(T')} c(e) = \sum_{e \in E(T) \setminus A} c(e) + \sum_{e \in B} c(e),$$

mientras que el coste del tour original es:

$$c(T) = \sum_{e \in E(T)} c(e) = \sum_{e \in E(T) \setminus A} c(e) + \sum_{e \in A} c(e).$$

Por lo tanto, la diferencia entre ambos costes es:

$$c(T) - c(T') = \sum_{e \in A} c(e) - \sum_{e \in B} c(e) = g(P).$$

Teniendo en cuenta que $g(P) > 0$, se concluye que:

$$c(T') < c(T),$$

que es lo que se quiere demostrar.

Ahora, veamos la segunda parte. Partiendo de P un camino alternante como en el enunciado, sea k el mayor índice para el cual la ganancia del camino que va desde x_0 hasta x_{2k} es mínima. A partir de esto, construimos un nuevo camino Q que empieza en x_{2k}, sigue hasta el final del camino P y luego continúa desde el inicio hasta volver a x_{2k}. Nuestro objetivo ahora es probar que este nuevo camino Q es propio. Primero, consideremos los subcaminos de Q que van desde x_{2k} hasta algún vértice entre x_{2k+1} y x_{2m}. Estos tramos corresponden a una parte del camino original P, por lo que su ganancia es la diferencia entre la ganancia de $(x_0, \ldots, x_{2i})$ y la de $(x_0, x_1, \ldots, x_{2k})$. Como k fue elegido de modo que esta última sea lo mínima posible, esta diferencia es necesariamente positiva. Ahora, consideramos los subcaminos de Q que van desde x_{2k} hasta algún vértice x_{2i}, con $i \leq k$. En este caso, la ganancia total se puede dividir en dos partes: la ganancia del tramo $(x_{2k}, \ldots, x_{2m})$, más la ganancia del tramo $(x_0, \ldots, x_{2i})$. Por definición de k, el último sumando es mayor o igual que la ganancia del tramo $(x_0, \ldots, x_{2k})$ y, entonces, la suma será mayor o igual que $g(P)$, que es mayor que cero. Por lo tanto, en ambos casos los subcaminos de Q tienen ganancia positiva, lo que prueba que efectivamente Q propio. □

Aunque en [14] se encuentra una formulación general del **algoritmo de Lin-Kernighan**, presentamos a continuación una variante más sencilla que generaliza al algoritmo *2-opt*, descrita con detalle en [11]. Antes de nada, es importante destacar que, si en un tour tenemos dos aristas (x_i, x_j), (x_k, x_l), involucrando a 4 nodos distintos, de forma que primero se recorre la primera arista y posteriormente la segunda, un *2-intercambio* consiste en reemplazarlas por las aristas (x_i, x_k) y (x_j, x_l).

Partiendo de un grafo con n nodos, se procede así:

① Seleccionar por cualquier procedimiento un tour T.

② Realizar el mejor 2-intercambio a partir de T, incluso aunque se empeore el coste y llamamos T^1 al nuevo tour. A continuación, realizar el mejor intercambio desde T^1, con la condición de no involucrar alguna arista añadida con anterioridad, obteniendo T^2. Repetimos la operación hasta que ya no quedan aristas del tour de partida y sea T' el tour con menor coste de $T^1, T^2, \ldots$.

③ Si $c(T') < c(T)$, volver al punto ② empezando con T'. En otro caso, **detenerse**.

Si se quiere ver una ilustración de la aplicación del heurístico general, obsérvese la Figura 4.3, presente en [14].

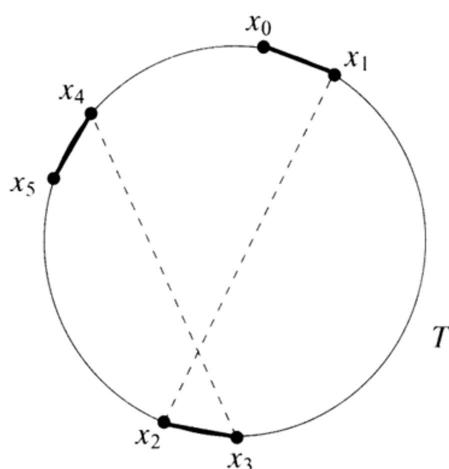

Figura 4.3: ejemplo de aplicación del algoritmo *Lin-Kernighan*.

Terminaremos este capítulo presentando otros algoritmos usados en la práctica.

4.1. Algoritmos de construcción de tours

En este apartado se tomará como referencia [1].

Algoritmo del vecino más cercano

Vamos a referirnos a este algoritmo como VC. La base de su funcionamiento es procurar acudir al nodo más cercano que aún no haya sido visitado. Para lograrlo, construimos un ordenamiento de las ciudades, de forma que la primera se elige arbitrariamente y a partir de ahí, se va escogiendo la más próxima a la última incorporada al tour entre las que no han sido visitadas. Una vez visitadas todas, desde la última se regresa a la ciudad de partida.

Si el número de ciudades es n, el tiempo de ejecución de VC es $O(n^2)$. Para un problema P con n ciudades cuyas distancias satisfacen la desigualdad triangular, en particular, se garantiza la propiedad que sigue.

$$\mathrm{VC}(P)/\mathrm{opt}(P) \leq 0{,}5(\lfloor \log_2 n \rfloor + 1),$$

donde $\mathrm{VC}(P)$ representa el coste del tour obtenido con el algoritmo VC y opt(P), el coste óptimo.

Sin embargo, no es posible una garantía sustancialmente mejor, ya que existen instancias donde la razón crece como $O(\log n)$ ([19]).

Algoritmo voraz

Algunos autores usan el término "*voraz*" para el *vecino más cercano*, pero es más apropiado reservarlo para la siguiente variante del "algoritmo *voraz*" de la teoría de matroides, que para el caso particular del TSP. Actúa del siguiente modo.

- Como siempre, consideramos una instancia del problema TSP como un grafo completo, G, de n vértices y cuyas aristas, e, poseen un coste $c(e)$. Entonces, podemos definir para cada par de nodos x_i y x_j, la distancia entre ellos: $c(x_i, x_j)$.
- Un tour es un ciclo hamiltoniano en este grafo (recordamos que un ciclo hamiltoninano es el camino que pasa exactamente **una vez** por cada nodo y termina en el mismo nodo en el que comenzó).

Para construir el ciclo hamiltoniano mediante este algoritmo, se procede de la siguiente forma.

Comenzamos con la arista más corta, es decir, aquella cuya distancia es la mínima: $\min \{c(x_i, x_k) : i \neq k\}$. Ahora, añadimos repetidamente la arista más corta entre las restantes. Una arista está disponible si no está aún en el tour, si su adición no crea un vértice de grado 3 y si su adición no forma un ciclo de longitud menor a n.

Cabe mencionar que este heurístico fue denominado “heurístico de multifragmento” por Bentley en [4] y [5].

El heurístico *voraz* puede implementarse para ejecutarse en tiempo $O(n^2\lceil\log_2 n\rceil)$, siendo algo más lento que el algoritmo del *vecino más cercano*. Sin embargo, la calidad de su tour en el peor caso puede ser algo mejor. Al igual que con VC, puede demostrarse que $\text{voraz}(P)/\text{opt}(P) \leq 0{,}5(\log_2 n + 1)$ para todos los problemas P que cumplen la desigualdad triangular ([16]), pero los peores ejemplos conocidos para el algoritmo *voraz* solo hacen que la razón crezca como $(\log n)/(3\log\log n)$ ([10]).

Algoritmo de Clarke y Wright

El heurístico de ahorros de *Clarke-Wright* (abreviadamente denotado por CW) se deriva de un algoritmo más general para enrutamiento de vehículos desarrollado en [7]. En el contexto del TSP la construcción es la que sigue.

Comenzamos con un *pseudotour* donde una ciudad elegida arbitrariamente actúa como centro, sea x_0. Tras ir a otra ciudad distinta de x_0, se vuelve a x_0. Equivalentemente, se construye un grafo donde cada vértice no-centro está conectado al centro por dos aristas, siendo una del camino de ida y otra del camino de vuelta. Para cada par de ciudades no-centro, definimos el *ahorro* así:

$$\delta(x_i, x_j) = c(x_i, x_0) + c(x_0, x_j) - c(x_i, x_j).$$

Representa la reducción en longitud si se fuese directamente de una ciudad a otra, evitando el centro.

Ahora se procede de modo análogo al algoritmo *voraz*. Ordenamos todos los pares no-centro por ahorros $\delta(x_i, x_j)$ de forma no creciente y vamos a través de ellos, evitando el centro siempre que no se cree un ciclo de vértices no centrales o siempre que un vértice no central se convierta en adyacente a más de otros dos vértices no centrales.

En el momento en el que sólo dos ciudades que no son un centro permanecen conectadas a dicho centro, el proceso se termina, pues en ese caso tendríamos un tour.

Al igual que en el caso del algoritmo *voraz*, este algoritmo puede ser implementado para actuar en tiempo $O(n^2 logn)$. También, si se cumple la desigualdad triangular, se garantiza la siguiente propiedad ([16]).

$$\frac{\text{CW}(P)}{\text{opt}(P)} \leq \lceil\log_2 n\rceil + 1,$$

donde $\text{CW}(P)$ es el coste del tour obtenido con el algoritmo CW aplicado al problema P con n ciudades. Mientras que en casos patológicos, su rendimiento es

$$\frac{\log n}{3\log\log n}$$

al igual que el obtenido en el algoritmo *voraz*.

Capítulo 5

Simulación del TSP

A continuación, con el objetivo de mostrar la aplicación práctica de los heurísticos estudiados de modo teórico, consideremos una aplicación real del problema del viajante de comercio.

Primeramente, se definirá con precisión el problema en cuestión. Luego, se resolverá empleando **AMPL**, un lenguaje de modelado de programación matemática y que permite hacer uso de solvers de problemas de programación entera mixta. De esta forma, se ofrecerá por un lado la función objetivo de la solución exacta del problema calculada por el solver. Además, mediante la variación del tamaño del problema, se compararán los tiempos de ejecución del programa y la influencia del tamaño n del problema considerado. Por último, se actuará de modo análogo pero esta vez en **R** y empleando el algoritmo de *Clarke-Wright* ya estudiado. Por lo cual, se analizarán minuciosamente las diferencias entre los resultados del solver y los del algoritmo expuesto.

5.1. Definición del problema

Somos jóvenes emprendedores de Santiago de Compostela y hemos creado una nueva empresa de fabricación e instalación de placas solares. Hemos desarrollado una innovadora tecnología que nos permite una eficiencia y almacenamiento de la energía nunca antes vistas. Para la comercialización de nuestro producto, hemos contratado a un comercial que se encarga de reunirse con otras empresas y particulares en las capitales de las distintas provincias de nuestro país. El problema radica en que sabiendo que tenemos que salir desde Santiago de Compostela, ¿cuál es la mejor ruta que pase por todas las capitales de provincia una sola vez y vuelva al inicio? Este es un problema típico de logística de una empresa.

Para intentar resolver este problema de modo óptimo, dado que poseemos amplios conocimientos en matemáticas, se nos ocurre asociar nuestra cuestión con un TSP. Para ello, tomaremos

como distancias las existentes entre las capitales de las provincias que deseemos visitar (tomaremos las distancias reales en km y aproximadas a números enteros mediante redondeo).

5.1.1. Galicia

Primeramente, debido a que nuestra empresa acaba de nacer y no hemos tenido una gran cantidad de beneficios, nos proponemos hacer un tour por las capitales de provincia gallegas. Por lo cual, tomamos como datos los siguientes.

- Número de ciudades: n=4.
- Distancias aproximadas entre las capitales de provincias dadas a través de una matriz de distancias.

Cuadro 5.1: Matriz de distancias entre ciudades gallegas (en km).

De / A	**Santiago**	**Pontevedra**	**Lugo**	**Ourense**
Santiago	0	63	98	106
Pontevedra	63	0	132	96
Lugo	98	132	0	123
Ourense	106	96	123	0

Cabe destacar que cuando empleemos AMPL, consideramos como valor de distancia $d(i,i) = 1000, \forall i \in \{1,2,3,4\}$ como una forma de penalización para evitar que se tome en el algoritmo el camino no coherente de una ciudad a sí misma.

Ahora, estamos en disposición de crear el archivo .mod para AMPL, que contiene el modelo del problema, incluyendo la definición de los parámetros del problema (número de ciudades y distancias entre ellas), las variables (binarias), la función objetivo y las restricciones, todo ello con la sintaxis requerida por AMPL.

```
param n;
param d{i in 1..n, j in 1..n};

var x{i in 1..n, j in 1..n} binary;
var u{i in 2..n}>=0;



minimize Ruta: sum{i in 1..n, j in 1..n}d[i,j]*x[i,j];
subject to salida {i in 1..n}:
```

```
sum{j in 1 ..n}x[i,j]=1;
subject to entrada {j in 1 ..n}:
sum{i in 1 ..n}x[i,j]=1;
subject to subciclo {i in 2 ..n, j in 2 ..n}:
u[i]-u[j]+n*x[i,j]<=n-1;
```

Podemos crear de la misma forma el archivo .dat adecuado, con los datos del problema a tratar.

```
data;
param n:= 4;
param d:      1      2      3      4:=
1      1000  63  98  106
2  63  1000  132  96
3  98  132  1000  123
4  106  96  123  1000;
```

Y el archivo .run, que contiene información sobre los ficheros que contienen el modelo y los datos, especifica el solver a utilizar (Gurobi en este caso) y las operaciones a realizar, típicamente resolver el problema y obtener el valor de la función objetivo y la solución óptima.

```
reset;
model modelo_tsp.mod;
data datos_tsp.dat;

option solver gurobi;
solve;

display x, Ruta;

display _total_solve_elapsed_time;
```

Con todo, obtenemos el siguiente resultado.

```
ampl: include comandos_tsp.run
Gurobi 9.5.2: optimal solution; objective 380
9 simplex iterations
1 branch-and-cut nodes
plus 2 simplex iterations for intbasis
x :=
1 1   0
1 2   0
1 3   1
1 4   0
```

```
2 1     1
2 2     0
2 3     0
2 4     0
3 1     0
3 2     0
3 3     0
3 4     1
4 1     0
4 2     1
4 3     0
4 4     0
;

        Ruta = 380

        _total_solve_elapsed_time = 1.328
```

Por un lado, hay que tener en cuenta que el solver utilizado es Gurobi 9.5.2 (lo que afectará a la velocidad de ejecución).

Observamos que el valor de la función objetivo respecto a la solución óptima calculada por el solver es 380, que nos da el tour siguiente: Santiago → Lugo → Ourense → Pontevedra → Santiago en el que se recorren, por tanto, 380 km.

Además, cabe destacar, que el tiempo de resolución es bastante pequeño (1.328 segundos) pues el número de parámetros es reducido (n=4).

Ahora haremos la resolución del problema propuesto, pero esta vez empleando el algoritmo *Clarke-Wright* programado en **R**. Cabe destacar que la función que hemos definido abarca un problema más general de rutas en el que además de pasar por cada nodo se transporta a cada uno una cantidad demandada de un cierto producto. En nuestro caso, definiremos una cantidad positiva arbitraria de capacidad del vehículo de transporte y consideraremos un vector de ceros como el vector de demandas, de forma que nos situaremos en un problema TSP simple. El algoritmo se muestra en el **Anexo I**.

Definiendo de modo conveniente la matriz de distancias se obtiene el siguiente resultado.

```
>capacidad.vehiculo<-10
>matriz.distancia<-matrix(c(0,63,98,106,63,0,132,96,98,
132,0,123,106,96,123,0),nrow=4)
>vector.demandas<-c(0,0,0,0)
>t=proc.time()
```

```
>Ahorrosalgorithm(vector.demandas,matriz.distancia,capacidad.vehiculo)
Tomamos las rutas :
0 1 3 2 0
Con un coste total de :
380
> proc.time()-t #tiempo de ejecucion
user  system elapsed
0       0       0
```

El valor de la función objetivo coincide con la anterior con un coste de 380. Sin embargo, la ruta hallada por el algoritmo de *Clarke-Wright* es un poco distinta: Santiago → Pontevedra → Ourense → Lugo → Santiago. Luego, dado que el coste total coincide con el de la solución óptima, este caso también es otra óptima posible. Además, se puede observar que el tiempo de ejecución es prácticamente nulo.

5.1.2. Noroeste de España

En este caso nos encontramos en la situación en la cual nuestra empresa ya ha crecido lo suficiente y, por lo tanto, nos planteamos extendernos por la zona noroeste de nuestro país, es decir, hacer un tour por las siguientes ciudades: Santiago (1), Pontevedra (2), Lugo (3), Ourense (4), Oviedo (5), Santander (6), León (7), Palencia (8) y Burgos (9). Este problema logístico más complejo posee los datos siguientes.

- Número de ciudades: n=9.
- Distancias aproximadas entre las capitales de provincias dadas a través de una matriz de distancias.

Cuadro 5.2: Matriz de distancias entre ciudades del noroeste (en km).

De / A	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	63	98	106	326	493	267	442	488
2	63	0	132	96	387	552	326	433	548
3	98	132	0	123	251	396	172	347	393
4	106	96	123	0	359	487	159	341	456
5	326	387	251	359	0	197	125	252	298
6	493	552	396	487	197	0	302	201	154
7	267	326	172	159	125	302	0	169	215
8	442	433	347	341	252	201	169	0	90
9	488	548	393	456	298	154	215	90	0

Ahora, de forma análoga a lo hecho anteriormente vamos a ver qué solución nos ofrece AMPL. Nótese que los archivos .run y .mod son iguales que en el caso anterior. El que si cambia es el archivo .dat.

```
#Datos del problema del viajante
data;
param n:= 9;
param d:1 2     3 4     5 6     7 8     9:=
1       1000  63  98  106  326  493  267  442  488
2       63  1000  132  96  387  552  326  433  548
3       98  132  1000  123  251  396  172  347  393
4       106  96  123  1000  359  487  159  341  456
5   326  387  251  359  1000  197  125  252   298
6   493  552  396  487  197  1000  302  201  154
7   267  326  172  159  125  302  1000  169  215
8   442  433  347  341  252  201  169  1000  90
9   488  548  393  456  298  154  215  90  1000
```

Como observamos, se cambian el número de ciudades y la matriz de distancias. Por lo tanto, se obtiene lo que vemos a continuación.

```
ampl: include comandos_tsp.run
Gurobi 9.5.2: optimal solution; objective 1277
98 simplex iterations
1 branch-and-cut nodes
plus 7 simplex iterations for intbasis
x [*,*]:
    1   2   3   4   5   6   7   8   9    :=
1   0   0   1   0   0   0   0   0   0
2   1   0   0   0   0   0   0   0   0
3   0   0   0   0   1   0   0   0   0
4   0   1   0   0   0   0   0   0   0
5   0   0   0   0   0   1   0   0   0
6   0   0   0   0   0   0   0   0   1
7   0   0   0   1   0   0   0   0   0
8   0   0   0   0   0   0   1   0   0
9   0   0   0   0   0   0   0   1   0
;

Ruta = 1277

_total_solve_elapsed_time = 1.437
```

La función objetivo de la solución óptima vemos que es 1277. Además, nos ofrece como tour óptimo el siguiente: Santiago → Lugo → Oviedo → Santander → Burgos → Palencia → León → Ourense → Pontevedra → Santiago en el que se recorren 1277km. También se puede apreciar que el tiempo de resolución es un poco superior al caso reducido (debido a que se requieren más iteraciones del algoritmo del símplex) pero sigue siendo pequeño (1.437 segundos).

Emplearemos de nuevo para este caso el algoritmo *Clarke-Wright* en R. Para ello, usaremos la misma función *Ahorrosalgorithm()* pero cambiando los datos correspondientes al problema actual.

```
capacidad.vehiculo<-10
>matriz.distancia<-matrix(c(0,63,98,106,326,493,267,442,488,63,0,
132,96,387,552,326,433,548,98,132,0,123,251,396,172,347,393,106,96,
123,0,359,487,159,341,456,326,387,251,359,0,197,125,252,298,493,
552,396,487,197,0,302,201,154,267,326,172,159,125,302,0,169,215,
442,433,347,341,252,201,169,0,90,488,548,393,456,298,154,215,90,
0),nrow=9)
> vector.demandas<-c(0,0,0,0,0,0,0,0,0)
> t=proc.time()
>Ahorrosalgorithm(vector.demandas,matriz.distancia,capacidad.vehiculo)
Tomamos las rutas :
0 2 4 5 8 7 6 3 1 0
Con un coste total de :
1277
> proc.time()-t #tiempo de ejecucion
user  system elapsed
0.00    0.01    0.02
```

Podemos observar que la ruta obtenida por el algoritmo *Clarke-Wright* coincide con la obtenida con programación entera: Santiago → Lugo → Oviedo → Santander → Burgos → Palencia → León → Ourense → Pontevedra → Santiago en el que se recorren también 1277km, pues es el valor de la función objetivo para ese tour. Además, el tiempo de ejecución es muy pequeño (0.02 segundos), pero superior al del caso reducido. Por lo cual, hemos visto en los dos ejemplos que el heurísticos de *Clarke-Wright* es más rápido que el algoritmo del símplex.

5.1.3. Norte de España

Por último, dado que nuestra empresa se encuentra en su máximo apogeo, buscamos que nuestro comercial se mueva por todas las ciudades (capitales de provincias) del norte del país, esto es, por las siguientes: Santiago (1), Pontevedra (2), Lugo (3), Ourense (4), Oviedo (5), Santander (6), Bilbao (7),Vitoria-Gasteiz (8), San Sebastián (9), Logroño (10), Pamplona (11),

Huesca (12), Zaragoza (13), Lleida (14), Girona (15), Barcelona (16), Tarragona (17), León (18), Palencia (19) y Burgos (20). Este es el problema logístico más complejo que vamos a tratar y posee las siguientes características.

- Número de ciudades: n=20.
- Distancias aproximadas entre las capitales de provincias dadas a través de una matriz de distancias.

Cuadro 5.3: Matriz de distancias entre ciudades del norte (en km).

De / A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	63	98	106	326	493	581	605	677	617	695	852	783	929	1142	1070	1000	267	442	488
2	63	0	132	96	387	552	641	684	738	696	774	931	862	1008	1202	1130	1060	326	433	548
3	98	132	0	123	251	396	506	510	602	521	600	757	688	834	1047	975	906	172	347	393
4	106	96	123	0	359	487	608	571	664	582	661	818	749	895	1112	1040	969	159	341	456
5	326	387	251	359	0	197	281	344	378	415	434	651	582	728	952	880	810	125	252	298
6	493	552	396	487	197	0	100	164	199	236	255	470	404	548	776	704	634	302	201	154
7	581	641	506	608	281	100	0	65	102	136	155	371	303	449	676	604	534	368	244	159
8	605	684	510	571	344	164	65	0	100	95	98	280	262	404	630	558	489	270	200	115
9	677	738	602	667	378	199	102	100	0	168	117	266	277	411	638	403	496	424	300	215
10	617	696	521	582	415	236	136	95	168	0	85	239	170	316	543	471	402	327	205	115
11	695	774	600	661	434	255	155	98	117	85	0	180	178	325	552	352	410	420	296	211
12	852	931	757	818	651	470	371	280	266	239	180	0	74	113	338	265	213	564	341	352
13	783	862	688	749	582	404	303	262	277	170	178	74	0	149	376	303	234	498	375	285
14	929	1008	834	895	728	548	449	404	411	316	325	113	149	0	225	152	101	643	521	431
15	1142	1202	1047	1112	952	776	676	630	638	543	552	338	376	225	0	102	188	873	750	657
16	1070	1130	975	1040	880	704	604	558	403	471	352	265	303	152	102	0	94	798	675	585
17	1000	1060	906	969	810	634	534	489	496	402	410	213	234	101	188	94	0	727	605	515
18	267	326	172	159	125	302	368	270	424	327	420	564	498	643	873	798	727	0	169	215
19	442	433	347	341	252	201	244	200	300	205	296	341	375	521	750	675	605	169	0	90
20	488	548	393	456	298	154	159	115	215	115	211	352	285	431	657	585	515	215	90	0

Para AMPL, aumentaremos el valor de penalización a 10000 dado que en este apartado se trabajará con distancias más grandes.

Estamos en disposición de volver a usar el programa AMPL para resolver este problema con más ciudades del viajante de comercio. Los archivos .run y .mod se mantienen invariantes y, al igual que antes, modificamos el archivo .dat.

```
#Datos del problema del viajante
data;
param n:= 20;
param d:        1 2     3 4     5 6     7 8     9 10 11 12 13 14 15 16
   17 18 19 20 :=
1       10000 63 98106  326 493 581 605 677 617  695 852 783 929  1142
   1070 1000 267 442 488
2       63  10000  132  96  387  552 641  684 738 696 774 931 862 1008
   1202 1130 1060 326 433 548
```

```
3       98 132 10000 123 251 396 506  510  602 521 600 757  688 834
    1047 975 906 172 347 393
4       106  96  123  10000  359  487 608  571  664 582  661   818 749
    895 1112 1040 969 159 341 456
5   326  387  251 359  10000 197 281  344  378 415  434 651 582 728 952
    880 810 125 252 298
6   493  552  396  487  197 10000 100  164  199 236  255 470 404  548
    776 704 634 302 201 154
7   581  641 506  608  281 100 10000 65   102 136 155
371 303 449 676 604 534 368 244 159
8   605  684  510  571  344  164  65 10000  100 95 98  280  262 404 630
     558 489 270 200 115
9   677  738 602  667 378   199  102  100 10000  168 117 266  277 411
    638 403 496 424 300 215
10  617 696  521  582  415  236  136  95  168 10000 85 239  170 316 543
    471 402 327 205 115
11  695  774 600  661 434 255 155  98  117 85 10000  180 178 325 552
    352 410 420 296 211
12  852 931  757  818  651  470  371  280  266 239 180  10000 74 113
    338 265 213 564 341 352
13  783 862  688 749   582  404  303 262  277 170 178   74 10000 149
    376 303 234  498 375 285
14  929  1008  834  895  728  548  449  404  411 316 325  113 149 10000
     225 152 101 643 521 431
15  1142 1202  1047   1112 952 776 676 630 638 543  552  338 376 225
    10000 102 188 873 750 657
16  1070   1130 975   1040  880 704  604  558 403 471  352 265 303  152
    102 10000  94 798 675 585
17  1000  1060   906  969  810 634   534  489 496 402  410  213 234 101
    188 94 10000 727 605 515
18  267  326  172  159  125 302 368  270 424 327 420  564 498 643 873
    798 727 10000 169 215
19  442 433  347   341  252  201  244 200 300 205 296  341 375 521 750
    675 605 169 10000 90
20  488 548 393   456 298  154  159 115  215 115 211 352 285 431 657
    585 515 215 90 10000;
```

Vemos que se cambian el número de ciudades n y la matriz de distancias. Por lo cual, obtenemos lo que sigue.

```
ampl: include comandos_tsp.run
Gurobi 9.5.2: optimal solution; objective 2720
55866 simplex iterations
```

```
9466 branch-and-cut nodes
plus 18 simplex iterations for intbasis
x [*,*]
:    1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
   18  19 :=
1    0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0
2    0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0
3    1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0
4    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     1   0
5    0   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0
6    0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0
     0   0
7    0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
     0   0
8    0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0
     0   0
9    0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0
     0   0
10   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0
     0   0
11   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0
     0   0
12   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0
     0   0
13   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0
     0   0
14   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1
     0   0
15   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0
     0   0
16   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0
     0   0
17   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0
     0   0
18   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   1
19   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
     0   0
```

```
20   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0
     0   0

:   20    :=
1    0
2    0
3    0
4    0
5    0
6    0
7    0
8    0
9    0
10   0
11   0
12   0
13   0
14   0
15   0
16   0
17   0
18   0
19   1
20   0
;

Ruta = 2720

_total_solve_elapsed_time = 2.219
```

Nótese que la función objetivo calculada por NEOS es de 2720. El tour óptimo por tanto es: Santiago → Lugo → Oviedo → Santander → Bilbao → Vitoria → San Sebastián → Pamplona → Barcelona → Girona → Tarragona → LLeida → Huesca → Zaragoza → Logroño → Burgos → Palencia → León → Ourense → Pontevedra → Santiago en el que se recorren 2720km. Además, el tiempo de ejecución es de igual modo relativamete pequeño (2.219 segundos) pero superior al de los anteriores casos.

Para concluir, emplearemos el heurístico de *Clarke-Wright* mediante R, con la función ya definida *Ahorrosalgorithm()* y con los datos correspondientes al actual problema.

```
> capacidad.vehiculo<-10
> matriz.distancia<-matrix(c(
+   0, 63, 98, 106, 326, 493, 581, 605, 677, 617, 695, 852, 783, 929,
```

```
   1142, 1070, 1000, 267, 442, 488,
+   63, 0, 132, 96, 387, 552, 641, 684, 738, 696, 774, 931, 862, 1008,
   1202, 1130, 1060, 326, 433, 548,
+   98, 132, 0, 123, 251, 396, 506, 510, 602, 521, 600, 757, 688, 834,
   1047, 975, 906, 172, 347, 393,
+   106, 96, 123, 0, 359, 487, 608, 571, 664, 582, 661, 818, 749, 895,
   1112, 1040, 969, 159, 341, 456,
+   326, 387, 251, 359, 0, 197, 281, 344, 378, 415, 434, 651, 582, 728,
   952, 880, 810, 125, 252, 298,
+   493, 552, 396, 487, 197, 0, 100, 164, 199, 236, 255, 470, 404, 548,
   776, 704, 634, 302, 201, 154,
+   581, 641, 506, 608, 281, 100, 0, 65, 102, 136, 155, 371, 303, 449,
   676, 604, 534, 368, 244, 159,
+   605, 684, 510, 571, 344, 164, 65, 0, 100, 95, 98, 280, 262, 404,
   630, 558, 489, 270, 200, 115,
+   677, 738, 602, 667, 378, 199, 102, 100, 0, 168, 117, 266, 277, 411,
   638, 403, 496, 424, 300, 215,
+   617, 696, 521, 582, 415, 236, 136, 95, 168, 0, 85, 239, 170, 316,
   543, 471, 402, 327, 205, 115,
+   695, 774, 600, 661, 434, 255, 155, 98, 117, 85, 0, 180, 178, 325,
   552, 352, 410, 420, 296, 211,
+   852, 931, 757, 818, 651, 470, 371, 280, 266, 239, 180, 0, 74, 113,
   338, 265, 213, 564, 341, 352,
+   783, 862, 688, 749, 582, 404, 303, 262, 277, 170, 178, 74, 0, 149,
   376, 303, 234, 498, 375, 285,
+   929, 1008, 834, 895, 728, 548, 449, 404, 411, 316, 325, 113, 149,
   0, 225, 152, 101, 643, 521, 431,
+   1142, 1202, 1047, 1112, 952, 776, 676, 630, 638, 543, 552, 338,
   376, 225, 0, 102, 188, 873, 750, 657,
+   1070, 1130, 975, 1040, 880, 704, 604, 558, 403, 471, 352, 265, 303,
   152, 102, 0, 94, 798, 675, 585,
+   1000, 1060, 906, 969, 810, 634, 534, 489, 496, 402, 410, 213, 234,
   101, 188, 94, 0, 727, 605, 515,
+   267, 326, 172, 159, 125, 302, 368, 270, 424, 327, 420, 564, 498,
   643, 873, 798, 727, 0, 169, 215,
+   442, 433, 347, 341, 252, 201, 244, 200, 300, 205, 296, 341, 375,
   521, 750, 675, 605, 169, 0, 90,
+   488, 548, 393, 456, 298, 154, 159, 115, 215, 115, 211, 352, 285,
   431, 657, 585, 515, 215, 90, 0
+ ),nrow=20)
> isSymmetric(matriz.distancia)
[1] TRUE
>vector.demandas<-c(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

```
> t=proc.time()
>Ahorrosalgorithm(vector.demandas,matriz.distancia,capacidad.vehiculo)
Tomamos las rutas :
0 1 3 17 18 19 9 16 15 14 13 11 12 10 8 7 6 5 4 2 0
Con un coste total de :
2808
> proc.time()-t #tiempo de ejecucion
user  system elapsed
0.11    0.01    0.09
```

Podemos apreciar una gran diferencia con los casos previos pues el valor del objetivo empleando el heurístico, 2808, es mayor que el mismo en la solución óptima, 2720. El camino dado por el heurístico es Santiago → Pontevedra → Ourense → León → Palencia → Burgos → Logroño → Tarragona → Barcelona → Girona → Lleida → Huesca → Zaragoza → Pamplona → San Sebastián → Vitoria → Bilbao → Santander → Oviedo → Lugo → Santiago con una distancia recorrida de 2808km, 88km más que en el caso óptimo. Sin embargo, el tiempo de ejecución es menor al caso anterior, con un valor de 0.09 segundos.

Con este ejemplo queda muy claro que a medida que aumentamos el número de datos (el número de ciudades) también lo hace la diferencia entre la solución óptima y la calculada por el heurístico de *Clarke-Wright*.

Otra cosa a tomar en cuenta es que en tamaños de datos medianos o pequeños (como en el caso n=20) no se nota mucho la diferencia de tiempo entre la resolución con el solver y la resolución con *Clarke-Wright* programado con R. Esto es debido a que Gurobi es potente y está muy bien optimizado (implementado en C++), por lo que actúa de modo muy eficiente en tamaños pequeños.

Los resultados obtenidos en este apartado se pueden sintetizar con la siguiente tabla.

Cuadro 5.4: Tabla resumen de la simulación.

Número de ciudades, n	Tiempo de AMPL (s)	Tiempo de CW (s)	Función Objetivo AMPL	Función objetivo CW
4	1.328	0	380	380
9	1.437	0.02	1277	1277
20	2.219	0.09	2720	2808

Otros métodos para analizar un TSP

Teniendo como objetivo completar el proceso de simulación/visualización del problema del viajante de comercio con n=20 (el más interesante), presentamos a continuación algunos mecanismos útiles.

- **Librería *TSP***. En primer lugar, podemos emplear la librería de R *TSP* que tiene implementados diversos algoritmos como 2-opt, vecino más cercano y otros que se basan en construir el tour a base de ir insertando nodos en el lugar más adecuado. Por lo tanto, creamos un script que albergue la matriz de distancias y actuamos del siguiente modo.

```
> numerodeciudades=20
> n=numerodeciudades
> M<-matrix(c(
+   0, 63, 98, 106, 326, 493, 581, 605, 677, 617, 695, 852, 783,
   929, 1142, 1070, 1000, 267, 442, 488,
+   63, 0, 132, 96, 387, 552, 641, 684, 738, 696, 774, 931, 862,
   1008, 1202, 1130, 1060, 326, 433, 548,
+   98, 132, 0, 123, 251, 396, 506, 510, 602, 521, 600, 757, 688,
   834, 1047, 975, 906, 172, 347, 393,
+   106, 96, 123, 0, 359, 487, 608, 571, 664, 582, 661, 818, 749,
   895, 1112, 1040, 969, 159, 341, 456,
+   326, 387, 251, 359, 0, 197, 281, 344, 378, 415, 434, 651, 582,
   728, 952, 880, 810, 125, 252, 298,
+   493, 552, 396, 487, 197, 0, 100, 164, 199, 236, 255, 470, 404,
   548, 776, 704, 634, 302, 201, 154,
+   581, 641, 506, 608, 281, 100, 0, 65, 102, 136, 155, 371, 303,
   449, 676, 604, 534, 368, 244, 159,
+   605, 684, 510, 571, 344, 164, 65, 0, 100, 95, 98, 280, 262,
   404, 630, 558, 489, 270, 200, 115,
+   677, 738, 602, 667, 378, 199, 102, 100, 0, 168, 117, 266, 277,
   411, 638, 403, 496, 424, 300, 215,
+   617, 696, 521, 582, 415, 236, 136, 95, 168, 0, 85, 239, 170,
   316, 543, 471, 402, 327, 205, 115,
+   695, 774, 600, 661, 434, 255, 155, 98, 117, 85, 0, 180, 178,
   325, 552, 352, 410, 420, 296, 211,
+   852, 931, 757, 818, 651, 470, 371, 280, 266, 239, 180, 0, 74,
   113, 338, 265, 213, 564, 341, 352,
+   783, 862, 688, 749, 582, 404, 303, 262, 277, 170, 178, 74, 0,
   149, 376, 303, 234, 498, 375, 285,
+   929, 1008, 834, 895, 728, 548, 449, 404, 411, 316, 325, 113,
   149, 0, 225, 152, 101, 643, 521, 431,
+   1142, 1202, 1047, 1112, 952, 776, 676, 630, 638, 543, 552,
   338, 376, 225, 0, 102, 188, 873, 750, 657,
+   1070, 1130, 975, 1040, 880, 704, 604, 558, 403, 471, 352, 265,
   303, 152, 102, 0, 94, 798, 675, 585,
+   1000, 1060, 906, 969, 810, 634, 534, 489, 496, 402, 410, 213,
   234, 101, 188, 94, 0, 727, 605, 515,
+   267, 326, 172, 159, 125, 302, 368, 270, 424, 327, 420, 564,
```

```
   498, 643, 873, 798, 727, 0, 169, 215,
+   442, 433, 347, 341, 252, 201, 244, 200, 300, 205, 296, 341,
   375, 521, 750, 675, 605, 169, 0, 90,
+   488, 548, 393, 456, 298, 154, 159, 115, 215, 115, 211, 352,
   285, 431, 657, 585, 515, 215, 90, 0
+ ),nrow=20)
> library(TSP)
> tps=TSP(M)
> viajante=solve_TSP(tps,"two_opt")
> viajante
object of class 'TOUR'
result␣of␣method␣'two_opt' for 20 cities
tour length: 3056
> viajante=solve_TSP(tps,"nearest_insertion")
> viajante
object of class 'TOUR'
result␣of␣method␣'nearest_insertion' for 20 cities
tour length: 2861
> viajante=solve_TSP(tps,"cheapest_insertion")
> viajante
object of class 'TOUR'
result␣of␣method␣'cheapest_insertion' for 20 cities
tour length: 2804
> viajante=solve_TSP(tps,"repetitive_nn") #nn-> vecino mas cercano
> viajante
object of class 'TOUR'
result␣of␣method␣'repetitive_nn' for 20 cities
tour length: 3278
> orden<-as.integer(viajante)
> orden
[1]  9  8  7  6 20 19 18  5  3  1  2  4 10 11 13 12 14 17 16 15
> tour_length(tps,orden)
[1] 3278
```

Como podemos apreciar, dependiendo del algoritmo que escojamos para la resolución del problema nos da una solución distinta. Por ejemplo, para el caso del *2-opt*, la función objetivo es 3056 mientras que para el caso del *vecino más cercano*, la función objetivo es 3278. Comparando estos valores con los obtenidos mediante el heurístico *Clarke-Wright*, nos percatamos de que para este caso el algoritmo *Clarke-Wright* es mejor que el algoritmo del *vecino más cercano* y que el algoritmo *2-opt* (pues la función objetivo valía 2808). Además, podemos fácilmente obtener el tour calculado por cualquiera de los algoritmos aplicando la función *as.integer*(viajante). No obstante, estos algoritmos están programados con una

componente estocástica, por lo que en el **Anexo II** mostramos el mejor resultado de 10 ejecuciones (para cada uno de ellos).

- **Templado simulado**. En este caso, seguiremos empleando R pero esta vez para aplicar el templado simulado. Es una heurística más sofisticada que las anteriores y que nos viene muy bien para comparar. Para obtener los resultados requeridos basta introducir la matriz de distancias correspondiente y ejecutar el algoritmo definido en [13]. Para visualizar el código y la aplicación a nuestro caso particular, mírese **Anexo III**. Cabe mencionar que, en este caso, el templado simulado alcanza la solución exacta.

- **Concorde**. Por último, en NEOS podemos emplear Concorde que es un solver de lo más potente para resolver TSP que hace uso de ramificación y acotación además de otras técnicas. Lo único que debemos hacer es crear un archivo .txt en formato TSPlib con la dimensión del problema y su matriz de distancias (véase **Anexo IV**). En consecuencia, utilizando NEOS tenemos el resultado siguiente.

```
/home/neos6/bin/concorde.cplex -s 99 -f sample.tsp
Host: sokrates  Current process id: 67169
Using random seed 99
Problem Name: Ejemplo matriz de distancias
Formato TSPlib para matriz de distancias (simetrica)
Problem Type: TSP
Number of Nodes: 20
Explicit Lengths (CC_MATRIXNORM)
Set initial upperbound to 2720 (from tour)
LP Value  1: 2454.000000  (0.00 seconds)
LP Value  2: 2720.000000  (0.00 seconds)
New lower bound: 2720.000000
Final lower bound 2720.000000, upper bound 2720.000000
Exact lower bound: 2720.000000
DIFF: 0.000000
Final LP has 27 rows, 46 columns, 121 nonzeros
Optimal Solution: 2720.00
Number of bbnodes: 1
Total Running Time: 0.01 (seconds)


***  ***




*** You chose the Concorde(CPLEX) solver ***
```

```
*** Cities are numbered 0..n-1 and each line shows a leg from one
    city to the next
followed by the distance rounded to integers***

20 20
0 2 98
2 4 251
4 5 197
5 6 100
6 7 65
7 8 100
8 10 117
10 15 352
15 14 102
14 16 188
16 13 101
13 11 113
11 12 74
12 9 170
9 19 115
19 18 90
18 17 169
17 3 159
3 1 96
1 0 63
```

Evidentemente, el tour resultante y la función objetivo coinciden con las ya calculadas anteriormente con Gurobi. Sin embargo, observamos que el tiempo de resolución, 0.01 segundos, es menor (recordemos que con Gurobi era de 2.047 segundos). Por lo que se aprecia la potencia para resolver TSP que comentábamos.

Anexo I

Código del algoritmo *Clarke-Wright*

```
Ahorrosalgorithm<-function(vector.demandas,matriz.distancia,
capacidad.vehiculo){
        n<-dim(matriz.distancia)[1] #numero de clientes mas deposito
        c<-numeric(n) #vector costes de rutas
        R<-matrix(0,nrow=n,ncol=3) #matriz de rutas
        ##########Paso 1: calcular rutas ir y volver
        c<-matriz.distancia[1,]*2 #Coste ir desde el deposito al
           cliente i y volver
        R[,2]<-1:n #Generamos las rutas (0,i,0) donde 0 es deposito
        R[1,2]<-0
        ctotal<-sum(c) #Coste total de ir desde cada deposito al cliente
        S<-matrix(0,nrow=n,ncol=n) #Matriz ahorros
        ############Paso 2: calcular los ahorros
        for(i in 2:n){
        for(j in 2:n){
        {
        if(i!=j){
        S[i,j]<-matriz.distancia[i,1]+matriz.distancia[1,j]-
        matriz.distancia[i,j]
        }
        }
        }
        }
        ###########Paso 3: optimizar rutas
        indicar<-1
        Sm<-1 #Valores de entrada del primer while
        while(Sm>0){ #Mientras existan ahorros mayores que cero buscamos
        rutas factibles
        Sm<-max(S) #Ahorro maximo
```

```
if(Sm>0){
{ #Posicion ahorro maximo
if(order(S,decreasing=TRUE)[1]%%n==0){
Positionfilas<-n
Positioncolumnas<-order(S,decreasing=TRUE)[1]%/%n
}
else{
Positionfilas<-order(S,decreasing=TRUE)[1]%%n
Positioncolumnas<-order(S,decreasing=TRUE)[1]%/%n + 1
}
}
#Demandas de los clientes i y j
CargaT<-vector.demandas[Positionfilas]+
vector.demandas[Positioncolumnas]
{ #Indicamos a que cliente visitamos antes de ir a i y despues
   de ir a j
if(R[Positionfilas,3]==0 && R[Positioncolumnas,1]==0
&& CargaT<=capacidad.vehiculo){
newPositionfilas<-Positionfilas
newPositioncolumnas<-Positioncolumnas
x<-0 #Evitamos ciclos
while(R[newPositionfilas,1]!=0){ #Sumamos la carga de los
   clientes
anteriores a i
CargaT<-CargaT+vector.demandas[R[newPositionfilas,1]]
newPositionfilas<-R[newPositionfilas,1]
if(newPositionfilas==Positioncolumnas) x<-x+1
}
while(R[newPositioncolumnas,3]!=0){ #Sumamos la carga de los
   clientes posteriores a j
CargaT<-CargaT+vector.demandas[R[newPositioncolumnas,3]]
newPositioncolumnas<-R[newPositioncolumnas,3]
if(newPositioncolumna==Positionfilas) x<-x+1
}
if(CargaT<=capacidad.vehiculo && x==0){ #Anadimos la ruta si es
   factible
R[Positionfilas,3]<-Positioncolumnas
R[Positioncolumnas,1]<-Positionfilas
}
S[Positionfilas,Positioncolumnas]<-0
S[Positioncolumnas,Positionfilas]<-0 #Borramos ahorros
   utilizados para evitar ciclos
}
```

```
        }
        S[Positionfilas,Positioncolumnas]<-0 #Si no es factible tambien
           lo borramos
        }
        } #Fin del while
        rutas<<-numeric() #Vector de rutas FINAL
        rutas[1]<<-0 #Empezamos en el deposito
        indicador<-2 #Nos movemos por el vector de rutas
        #Creamos la ruta final
        for(i in 2:n){
        if(R[i,1]==0){
        rutas[indicador]<<-i
        while(rutas[indicador]!=0){
        rutas[indicador+1]<<-R[rutas[indicador],3]
        indicador<-indicador+1
        }
        indicador<-indicador+1
        }
        }
        rutas[which(rutas==0)]<-1
        coste.total<-0
        for(i in 1:(length(rutas)-1)){
        coste.total<-coste.total+matriz.distancia[rutas[i],rutas[i+1]]
        }
        rutas<-rutas-1
        cat("Tomamos␣las␣rutas",":\n")
        cat(rutas,"\n")
        cat("Con␣un␣coste␣total␣de",":\n")
        cat(coste.total,"\n")
} #Fin de la funcion
```

Anexo II

10 ejecuciones de los algoritmos de la librería *TSP*

```
>library(TSP)
>
> metodos <- c("two_opt", "nearest_insertion", "cheapest_insertion",
   "repetitive_nn")
>
> for (metodo in metodos) {
        +   cat("\nMetodo:", metodo, "\n")
        +   valores <- numeric(10)
        +   for (i in 1:10) {
                +     viajante <- solve_TSP(tps, method = metodo)
                +     valores[i] <- tour_length(viajante)
                +   }
        +   print(valores)
        +   cat("Minimo␣valor␣para", metodo, ":", min(valores), "\n")
        + }

Metodo: two_opt
[1] 2870 3457 2738 3010 3449 3092 2988 2901 3153 2926
Minimo valor para two_opt : 2738

Metodo: nearest_insertion
[1] 3036 2790 2738 2861 2861 2861 2946 3036 2738 2738
Minimo valor para nearest_insertion : 2738

Metodo: cheapest_insertion
[1] 2874 2804 2804 2804 2804 2738 2738 2738 2738 2738
```

```
Minimo valor para cheapest_insertion : 2738

Metodo: repetitive_nn
[1] 3278 3265 3278 3265 3278 3265 3265 3278 3265 3278
Minimo valor para repetitive_nn : 3265
```

Anexo III

Código y solución del templado simulado

```
>#Definicion de la matriz de coste
> coste<-matrix(c(
+   0, 63, 98, 106, 326, 493, 581, 605, 677, 617, 695, 852, 783, 929,
   1142, 1070, 1000, 267, 442, 488,
+   63, 0, 132, 96, 387, 552, 641, 684, 738, 696, 774, 931, 862, 1008,
   1202, 1130, 1060, 326, 433, 548,
+   98, 132, 0, 123, 251, 396, 506, 510, 602, 521, 600, 757, 688, 834,
   1047, 975, 906, 172, 347, 393,
+   106, 96, 123, 0, 359, 487, 608, 571, 664, 582, 661, 818, 749, 895,
   1112, 1040, 969, 159, 341, 456,
+   326, 387, 251, 359, 0, 197, 281, 344, 378, 415, 434, 651, 582, 728,
   952, 880, 810, 125, 252, 298,
+   493, 552, 396, 487, 197, 0, 100, 164, 199, 236, 255, 470, 404, 548,
   776, 704, 634, 302, 201, 154,
+   581, 641, 506, 608, 281, 100, 0, 65, 102, 136, 155, 371, 303, 449,
   676, 604, 534, 368, 244, 159,
+   605, 684, 510, 571, 344, 164, 65, 0, 100, 95, 98, 280, 262, 404,
   630, 558, 489, 270, 200, 115,
+   677, 738, 602, 664, 378, 199, 102, 100, 0, 168, 117, 266, 277, 411,
   638, 403, 496, 424, 300, 215,
+   617, 696, 521, 582, 415, 236, 136, 95, 168, 0, 85, 239, 170, 316,
   543, 471, 402, 327, 205, 115,
+   695, 774, 600, 661, 434, 255, 155, 98, 117, 85, 0, 180, 178, 325,
   552, 352, 410, 420, 296, 211,
+   852, 931, 757, 818, 651, 470, 371, 280, 266, 239, 180, 0, 74, 113,
   338, 265, 213, 564, 341, 352,
```

```
+   783, 862, 688, 749, 582, 404, 303, 262, 277, 170, 178, 74, 0, 149,
   376, 303, 234, 498, 375, 285,
+   929, 1008, 834, 895, 728, 548, 449, 404, 411, 316, 325, 113, 149,
   0, 225, 152, 101, 643, 521, 431,
+   1142, 1202, 1047, 1112, 952, 776, 676, 630, 638, 543, 552, 338,
   376, 225, 0, 102, 188, 873, 750, 657,
+   1070, 1130, 975, 1040, 880, 704, 604, 558, 403, 471, 352, 265, 303,
   152, 102, 0, 94, 798, 675, 585,
+   1000, 1060, 906, 969, 810, 634, 534, 489, 496, 402, 410, 213, 234,
   101, 188, 94, 0, 727, 605, 515,
+   267, 326, 172, 159, 125, 302, 368, 270, 424, 327, 420, 564, 498,
   643, 873, 798, 727, 0, 169, 215,
+   442, 433, 347, 341, 252, 201, 244, 200, 300, 205, 296, 341, 375,
   521, 750, 675, 605, 169, 0, 90,
+   488, 548, 393, 456, 298, 154, 159, 115, 215, 115, 211, 352, 285,
   431, 657, 585, 515, 215, 90, 0
+ ),nrow=20,byrow=T)
>
> #Solucion inicial
> sol0<-c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1)
> itmax<-10
> ini<-proc.time()[1]
> funcion_obj<-function(x,coste){
+   cost<-0
+   for (i in 1:(length(x)-1)){
+     cost<-cost+coste[x[i],x[i+1]]
+   }
+   return(cost)}
> Zc<-funcion_obj(sol0,coste)
> Salida<-c(0,sol0,Zc)
> for(i in 1:itmax){
+   if(i==1){T=0.2*Zc}
+   if(i>1){T=0.5*T}
+   for(j in 1:itmax){
+     camb=0
+     while(camb==0){
+       n=(length(sol0)-2)
+       while(n == (length(sol0)-2)) {
+         a<-sample(2:(length(sol0)-2),1)
+         if(a==(length(sol0)-2)){b<-(length(sol0)-1)}
+         if(a!=(length(sol0)-2)){b<-sample((a+1):(length(sol0)-1),1)}
+         n=b-a}
+       solnew<-c(sol0[1:(a-1)],rev(sol0[a:b]),sol0[(b+1):length(sol0)])
```

```
+         Zn<-funcion_obj(solnew,coste)
+
+         if(Zn<=Zc & is.finite(Zn)){
+           sol0=solnew
+           Zc=Zn
+           camb=1
+           Salida<-rbind(Salida,c(T,sol0,Zc))}
+         if(Zn>Zc & is.finite(Zn)){
+           P<-exp((Zc-Zn)/T)
+           selecc<-sample(c(0,1), 1, prob = c((1-P),P))
+           if (selecc==1){
+             sol0=solnew
+             Zc=Zn
+             camb=1
+             Salida<-rbind(Salida,c(T,sol0,Zc))}
+         }
+       }
+     }
+ }
> colnames(Salida)<-
    c("T","Nodo_Origen","Visita_1","Visita_2","Visita_3","Visita_4",
"Visita_5","Visita_6","Visita_7","Visita_8","Visita_9","Visita_10",
"Visita_11","Visita_12","Visita_13","Visita_14","Visita_15","Visita_16",
"Visita_17","Visita_18","Visita_19","Nodo_Origen","Coste")
> Salida
T Nodo_Origen Visita_1 Visita_2 Visita_3 Visita_4 Visita_5 Visita_6
    Visita_7 Visita_8 Visita_9 Visita_10
Salida    0.000000           1        2        3        4        5
          6        7        8        9       10       11
738.000000           1        2        3        4        5        6
          7        8        9       10       11
738.000000           1        2        3        4        6        5
          7        8        9       10       11
738.000000           1        2        3        4        6        5
          7        8        9       10       11
738.000000           1        2        3        4        6        5
          7        8        9       10       11
738.000000           1        2        3        4        6        5
          7       17       16       15       14
738.000000           1        2        3        4        6        5
          7       17       16       15       14
738.000000           1        2        3        4        6        5
          7       10        9        8       14
```

```
738.000000          1         2         3         4         6         5
         7        10         9         8        14
738.000000          1         2         3         4         6         5
         7        10         9         8        14
738.000000          1         2         3         4         6         5
         7        10         9         8        15
369.000000          1         2         4         3         6         5
         7        10         9         8        15
369.000000          1         2         4         3         6         5
         7        10         9         8        15
369.000000          1         2         4         3         6         5
         7        16        14        15         8
369.000000          1         2         4         3         6         5
         9         8        15        14        16
369.000000          1         2         4         3         6         5
         9         8        15        14        13
369.000000          1         2         4         3         6         5
        11        16         7        10        17
369.000000          1         2         4         3         6         5
        11        16         7        10        17
369.000000          1         2         4         3         6         5
        19        13        14        15         8
369.000000          1         2         4         3         6        10
        17        20        12         9         8
369.000000          1         2         4         3         6        10
        17        20        12         9         8
184.500000          1         2         4         3         6        10
        17        20        12         9         8
184.500000          1         2         4         3        16        18
        11         7         5        19        13
184.500000          1         2         4         3         6        10
        17        20        12         9         8
184.500000          1         2         4         3         6        10
        17        20        12         9         8
184.500000          1         2         4         3         6        10
        17        20        12         9         8
184.500000          1         2         4         3         6        10
        17        14         8         9        12
184.500000          1         2         4         3         6        10
        17        14         8         9        12
184.500000          1         2         4         3         6        10
        17        14         8         9        12
```

```
184.500000          1         2         4         3        16        11
         18         7         5        19        13
184.500000          1         2         4         3        16        11
         18         7         5        19        14
92.250000          1         2         4         3        16        11
         18         7         5        19        12
92.250000          1         2         4         3        16        11
         18         7         5        19        12
92.250000          1         2         4         3        16        17
         14        20        15        13         8
92.250000          1         2         4         3        16        17
         14        20        15        13         8
92.250000          1         2         4         3        16        17
         14        20        15        13         8
92.250000          1         2         4         3        16        17
         14        20        15        13         8
92.250000          1         2         4         3        16        17
         14        20        15        13         8
92.250000          1         2         4         3        16        17
         14        20         8        13        15
92.250000          1         2         4         3        16        17
         14        20         8        13        15
92.250000          1         2         4         3        16        17
         14        19        12         6         7
46.125000          1         2         4         3        16        17
         15         9        11        10         7
46.125000          1         2         4         3        16        17
         15         9        11        10         7
Visita_11 Visita_12 Visita_13 Visita_14 Visita_15 Visita_16 Visita_17
   Visita_18 Visita_19 Nodo_Origen Coste
Salida         12         13         14         15         16         17
          18         19         20            1  3690
12         13         14         15         16         17         20
   19         18            1  3257
12         13         14         15         16         17         20
   19         18            1  3566
12         13         14         15         16         17         19
   20         18            1  3702
12         13         14         15         16         17         19
   18         20            1  4002
13         12         11         10          9          8         19
   18         20            1  4066
```

```
8           9          10          11          12          13          19          18
            20             1   4496
15          16          17          11          12          13          19
    18          20             1   4423
15          16          17          11          12          13          19
    20          18             1   4123
15          16          17          20          19          13          12
    11          18             1   4433
14          16          17          20          19          13          12
    11          18             1   4709
14          16          17          20          19          13          12
    11          18             1   4582
14          16          17          20          19          13          11
    12          18             1   4830
9          10          17          20          19          13          11          12
            18             1   5606
7          10          17          20          19          13          11          12
            18             1   5671
19          20          17          10           7          16          11
    12          18             1   5842
20          19          13          14          15           8           9
    12          18             1   5984
20          12           9           8          15          14          13
    19          18             1   5851
9          12          20          17          10           7          16          11
            18             1   5920
15          14          13          19           5           7          16
    11          18             1   6104
15          14          13          19           5           7          16
    18          11             1   6978
15          14          13          19           5           7          11
    18          16             1   6904
14          15           8           9          12          20          17
    10           6             1   6906
15          14          13          19           5           7          11
    18          16             1   6904
14          15          13          19           5           7          11
    18          16             1   6905
14          15          13          19           5           7          18
    11          16             1   6672
20          15          13          19           5           7          18
    11          16             1   6690
```

```
20          15          13          19          5          7          18
    16          11              1   6693
20          15          13          19          5          7          18
    11          16              1   6690
15          20          12           9          8         14          17
    10           6              1   6692
8           9          12          20          15         13          17          10
            6              1   6971
9           8          14          20          15         13          17          10
            6              1   6870
9           8          13          15          20         14          17          10
            6              1   6595
9          12          19           5           7         18          11          10
            6              1   6020
9          10          11          18           7          5          19          12
            6              1   6156
9          10          11          18           7          6          12          19
            5              1   5808
9          10          11          18          19         12           6           7
            5              1   5638
9          11          10          18          19         12           6           7
            5              1   5494
9          11          10          18          19         12           6           7
            5              1   5490
9          11          10           7           6         12          19          18
            5              1   5143
10          11           9          15          13          8          20
    18           5              1   5279
6          12          19          14          13          8          20          18
            5              1   5139
6          12          19          14          13          8          20           5
           18              1   5163
[ reached getOption("max.print") -- omitted 58 rows ]
> fin=proc.time()[1]
> ini
user.self
21.93
> fin-ini
user.self
0.35
```

Anexo IV

Archivo para Concorde

```
NAME : Ejemplo matriz de distancias
COMMENT : Formato TSPlib para matriz de distancias (simetrica)
TYPE : TSP
DIMENSION : 20
EDGE_WEIGHT_TYPE : EXPLICIT
EDGE_WEIGHT_FORMAT : FULL_MATRIX
EDGE_WEIGHT_SECTION :
0 63 98 106 326 493 581 605 677 617 695 852 783 929 1142 1070 1000 267
    442 488
63 0 132 96 387 552 641 684 738 696 774 931 862 1008 1202 1130 1060 326
    433 548
98 132 0 123 251 396 506 510 602 521 600 757 688 834 1047 975 906 172
    347 393
106 96 123 0 359 487 608 571 664 582 661 818 749 895 1112 1040 969 159
    341 456
326 387 251 359 0 197 281 344 378 415 434 651 582 728 952 880 810 125
    252 298
493 552 396 487 197 0 100 164 199 236 255 470 404 548 776 704 634 302
    201 154
581 641 506 608 281 100 0 65 102 136 155 371 303 449 676 604 534 368
    244 159
605 684 510 571 344 164 65 0 100 95 98 280 262 404 630 558 489 270 200
    115
677 738 602 664 378 199 102 100 0 168 117 266 277 411 638 403 496 424
    300 215
617 696 521 582 415 236 136 95 168 0 85 239 170 316 543 471 402 327 205
    115
695 774 600 661 434 255 155 98 117 85 0 180 178 325 552 352 410 420 296
    211
```

```
852 931 757 818 651 470 371 280 266 239 180 0 74 113 338 265 213 564
    341 352
783 862 688 749 582 404 303 262 277 170 178 74 0 149 376 303 234 498
    375 285
929 1008 834 895 728 548 449 404 411 316 325 113 149 0 225 152 101 643
    521 431
1142 1202 1047 1112 952 776 676 630 638 543 552 338 376 225 0 102 188
    873 750 657
1070 1130 975 1040 880 704 604 558 403 471 352 265 303 152 102 0 94 798
    675 585
1000 1060 906 969 810 634 534 489 496 402 410 213 234 101 188 94 0 727
    605 515
267 326 172 159 125 302 368 270 424 327 420 564 498 643 873 798 727 0
    169 215
442 433 347 341 252 201 244 200 300 205 296 341 375 521 750 675 605 169
    0 90
488 548 393 456 298 154 159 115 215 115 211 352 285 431 657 585 515 215
    90 0
EOF
```

Bibliografía

[1] Aarts, E. y Lenstra, J. K. (1997). *Local search in combinatorial optimization.* Princeton University Press, Princeton, NJ, USA.

[2] Bazaraa, M. S., Jarvis, J. J. y Sherali, H. D. (2009). *Linear programming and network flows.* Wiley, Hoboken, NJ, USA.

[3] Bellman, R. (1956-1957). *Dynamic Programming*, Princeton University Press, Princeton, NJ, USA.

[4] Bentley, J. J. (1990a). *Experiments on traveling salesman heuristics*, Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 91-99, San Francisco, CA, USA.

[5] Bentley, J. J. (1992). *Fast algorithms for geometric traveling salesman problems*, ORSA Journal on Computing, **4**(4), 387-411, Baltimore, MD, USA.

[6] Casas Méndez, B., Gómez Casares, I. y González Díaz, J. (2023). *Material docente de Programación Lineal y Entera*, Curso 2022–2023, Facultad de Matemáticas, Universidad de Santiago de Compostela.

[7] Clarke, G. y Wright, J. W. (1964). *Scheduling of vehicles from a central depot to a number of delivery points*, Operations Research, **12**(4), 568-581, Baltimore, MD, USA.

[8] Dantzig, G. B. (1957). *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, USA.

[9] Ford, L. R. y Fulkerson, D. R. (1956). *Maximal flow through a network*, Canadian Journal of Mathematics, **8**, 399-404, Ottawa, Canada.

[10] Frieze, A. M. (1979). *Worst-case analysis of algorithm for the travelling salesman problem*, SIAM Journal on Computing, **8**(4), 521-529, Philadelphia, PA, USA.

[11] González Díaz, J. (2012) *Técnicas de Optimización de la Gestión.* Manuscrito.

[12] Johnson, D. S., Aho, A. V., Hopcroft, J. E. y Ullman, J. D. (1988). *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, USA.

[13] Kirkpatrick, S., Gelatt, C. D. y Vecchi, M. P. (1983). *Optimization by Simulated Annealing.* Science, **220**(4598), 671-680.

[14] Korte, B. y Vygen, J. (2001). *Combinatorial optimization: Theory and algorithms*, 2nd ed. Springer, Berlin, Germany.

[15] Lin, S. y Kernighan, B. W. (1973). *An effective heuristic algorithm for the traveling-salesman problem*, Operations Research, **21**(2), 498–516, Baltimore, MD, USA.

[16] Ong, H. L. y Moore, J. B. (1984). *Worst-case analysis of two travelling salesman heuristics*, Operations Research Letters, **2**(6), 273-277, Amsterdam, Netherlands.

[17] Papadimitriou, C. H. (1992). *The complexity of the Lin-Kernighan heuristic for the traveling salesman problem*, SIAM Journal on Computing, **21**(3), 450-465, Philadelphia, PA, USA.

[18] Pisinger, D. (1999). *Core problems in Knapsack Algorithms*, Operations Research, **47**(4), 570-575, Linthicum, MD, USA.

[19] Rosenkrantz, D. J., Stearns, R. E. y Lewis, P. M. (1977). *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal on Computing, **6**(3), 563–581, Philadelphia, PA, USA.