UNIVERSIDAD DE SALAMANCA

FACULTAD DE CIENCIAS

DEPARTAMENTO DE ESTADÍSTICA

TRABAJO DE FIN DE GRADO

ESTUDIO DE ALGORITMOS PARA EL PROBLEMA DEL VIAJANTE: MODELIZACIÓN Y COMPARACIÓN DEL RENDIMIENTO

Tutor: Miguel Rodríguez Rosa

Autora: Paula Orizaola Molinero

Grado en Estadística

Curso académico 2023-24

UNIVERSIDAD DE SALAMANCA

FACULTAD DE CIENCIAS

DEPARTAMENTO DE ESTADÍSTICA

TRABAJO DE FIN DE GRADO

ESTUDIO DE ALGORITMOS PARA EL PROBLEMA DEL VIJANTE: MODELIZACIÓN Y COMPARACIÓN DEL RENDIMIENTO

Firmado por:

Autora: Paula Orizaola Molinero

Tutor: Miguel Rodríguez Rosa

Grado en Estadística

Curso académico 2023-24

VNiVERSiDAD D SALAMANCA
CAMPUS DE EXCELENCIA INTERNACIONAL

Facvltad D Ciencias
VNiVERSiDAD D SALAMANCA

Certificado del tutor TFG Grado en Estadística

D. Miguel Rodríguez Rosa, profesor del Departamento de Estadística de la Universidad de Salamanca,

HACE CONSTAR:

Que el trabajo titulado "*Estudios de algoritmos para el Problema del Viajante: Modelización y comparación del rendimiento*", que se presenta, ha sido realizado por D.ª Paula Orizaola Molinero, con DNI 45575306R y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Grado en Estadística en esta Universidad.

Salamanca, a 2 de julio de 2024.

RODRIGUEZ ROSA MIGUEL - 70821913Z

Firmado digitalmente por RODRIGUEZ ROSA MIGUEL - 70821913Z
Fecha: 2024.07.02 17:51:35 +02'00'

Fdo.: Miguel Rodríguez Rosa

Índice general

Resumen

El Problema del Viajante —Traveling Salesman Problem, TSP— es un desafío computacional (problema NP-hard) bien conocido y fundamental en la teoría de la optimización combinatoria. Consiste en encontrar la ruta más corta que visite un conjunto de ciudades exactamente una vez y regrese al punto de partida. El TSP tiene aplicaciones en una amplia gama de campos, desde logística y planificación de rutas hasta diseño de circuitos y bioinformática.

Este trabajo tiene como objetivo principal investigar y analizar diferentes algoritmos para resolver el TSP, tanto aquellos que buscan soluciones óptimas (algoritmos exactos) como los enfoques heurísticos que buscan soluciones aproximadas. Se explorarán algoritmos exactos, como el algoritmo de ramificación y poda, y heurísticos populares, como el algoritmo de colonia de hormigas. Se llevará a cabo un análisis exhaustivo de los algoritmos seleccionados, evaluando su rendimiento y complejidad computacional en función de diferentes instancias del problema y tamaños de redes. Además, se desarrollará un programa en el entorno de programación R para implementar y ejecutar estos algoritmos en ejemplos concretos.

Summary

The Traveling Salesman Problem (TSP) is a well-known and fundamental computational challenge (NP-hard problem) in the theory of combinatorial optimization. It involves finding the shortest route that visits a set of cities exactly once and returns to the starting point. The TSP has applications in a wide range of fields, from logistics and route planning to circuit design and bioinformatics.

This work aims to investigate and analyze different algorithms to solve the TSP, including those seeking optimal solutions (exact algorithms) as well as heuristic approaches aiming for approximate solutions. Exact algorithms such as the branch and bound algorithm will be explored, alongside popular heuristics like the ant colony algorithm. An exhaustive analysis of the selected algorithms will be conducted, evaluating their performance and computational complexity based on different instances of the problem and network sizes. Additionally, a program will be developed in the R programming environment to implement and execute these algorithms on specific examples.

Introducción

El Problema del Viajante surge como un desafío en forma de problema dentro de la optimización combinatoria. Este enigma, aparentemente simple, plantea lo siguiente: ¿Cuál es la ruta más corta que un viajante puede tomar para visitar un conjunto n de ciudades teniendo en cuenta que el punto de inicio y final debe ser el mismo?

Desde la primera vez que se formuló, en la década de 1930, el Problema del Viajante ha sido objeto de una intensa investigación, ya que lo podemos encontrar en diversas aplicaciones como la logística, la planificación de rutas, la genética computacional y la teoría de la complejidad computacional. No existe una solución general al problema, sí soluciones aproximadas

A pesar de que el Problema del Viajante no tiene un origen claro, se sabe que sobre 1830 aparece en una guía para viajantes en Alemania en la que nos encontramos ejemplos explicados, pero sin apenas formulación matemática (Voigt, 1831).
El problema se empezó a desarrollar en 1800 por dos matemáticos: el irlandés W.R. Hamilton y el británico Thomas Kirkman. Fue mediante un juego llamado Juego Icosiano de Hamilton, que surgió en 1857 (Ball & Coxeter, 1987) gracias a William Rowan Hamilton. La premisa era trazar una ruta a lo largo de las aristas de un dodecaedro, visitando cada vértice exactamente una vez y regresando al punto de partida. Este juego se volvió muy popular, y la gente compraba la revista donde aparecía con el fin de poder resolverlo, a modo de pasatiempo.

La formulación más general del TSP fue explorada por primera vez en la década de 1930 por matemáticos en Viena y Harvard, especialmente por Crilly (2005).

Menger no solo definió el problema, sino que también examinó el algoritmo de fuerza bruta y señaló las limitaciones de la heurística del vecino más cercano. Esta problemática se erige como un desafío NP-hard, concepto que veremos más tarde, en la optimización combinatoria, vital en la investigación operativa y las ciencias informáticas.

El Problema del Viajante fue popularizado por Merrill Flood, quien compartió su interés con Tucker en 1937. Merrill Flood Meeks fue un matemático estadounidense, conocido por sus aportaciones en el desarrollo de la teoría de juegos, en concreto por el famoso dilema del prisionero, junto con Melvin Dresher, que planteó la base del modelo de cooperación y conflicto del problema. Flood junto con Tucker, Miller y Zemlin idearon la primera interpretación matemática del algoritmo, la cual se denomina Formulación MTZ (González-Santander, 2020).

El TSP se puede resolver como un modelo de programación entera, y la formulación de Miller, Tucker y Zamlin es la siguiente:

Supongamos que tenemos n ciudades numeradas de 1 a n.

Sea x_{ij} una variable binaria que indica si el viajante va directamente de la ciudad i a la ciudad j ($x_{ij} = 1$ si el viajante va de la ciudad i a la ciudad j, $x_{ij} = 0$ de lo contrario).

También definimos u_i como variables continuas que representan la posición en el orden de visita

de cada ciudad.

Función objetivo:

$$\text{mín} \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} c_{ij} \cdot x_{ij}$$

Donde c_{ij} es el costo de viajar de la ciudad i a la ciudad j.

Teniendo en cuenta las siguientes restricciones:

Cada ciudad es visitada exactamente una vez:

$$\sum_{i=1, i\neq j}^{n} x_{ij} = 1 \quad \forall j \in \{1, 2, ..., n\}$$

$$\sum_{j=1, j\neq i}^{n} x_{ij} = 1 \quad \forall i \in \{1, 2, ..., n\}$$

Tan solo una ruta cruza los nodos:

$$u_i - u_j + n \cdot x_{ij} \leq n - 1 \quad \forall i, j \in \{2, 3, ..., n\}, i \neq j$$

$$u_i \geq 0 \quad \forall i \in \{2, 3, ..., n\}$$

$$u_1 = 0$$

Esta formulación asegura que el viajante visite cada ciudad exactamente una vez y que no haya subciclos en la solución (Flood & Savage, 1948).

Fue más tarde cuando la revista "Journal of the Operations Research Society of America" publicó el artículo "Soluciones de un problema del viajante de gran tamaño" de Dantzig et al. (2003), cuando la optimización pasó a un primer plano e impulsó el estudio de la combinatoria y la programación lineal, especialmente en problemas de transporte y asignación.

En la formulación propuesta por Dantzig, Fulkerson y Johnson partimos de las mismas variables que en la formulación MTZ, pero estos decidieron añadir una nueva restricción:

$$\sum_{i\in S} \sum_{j\in S} x_{ij} \leq (|S| - 1) \quad \forall S \subseteq V, S \neq \emptyset$$

Con esta nueva restricción, unida a las anteriores lograron eliminar los sub-recorridos, la solución da un recorrido único.

En la década de 1940, George Dantzig también desarrolló el método simplex como una solución eficiente para problemas de programación lineal.

Este método revolucionario permitió resolver problemas de optimización en los que se buscaba maximizar o minimizar una función lineal sujeta a ciertas restricciones lineales. El método simplex opera moviéndose de vértice a vértice en un poliedro convexo, donde cada vértice representa una solución del problema. Utilizando la estructura matemática del poliedro, el algoritmo simplex busca mejorar la solución a través de movimientos hacia vértices adyacentes. El proceso iterativo continúa hasta alcanzar una solución óptima o determinar que el problema es no acotado. El método simplex se convirtió en una herramienta fundamental para la optimización en una amplia gama de campos (Dantzig et al., 1955).

En 1954 Dantzig, Fulkerson y Johnson introdujeron su método para resolver el problema del horario de los buques (Dantzig et al., 2003).

Este problema involucra la planificación eficiente de rutas y horarios para una flota de buques, considerando restricciones como la capacidad de carga, tiempos de tránsito, costos operativos y demanda de transporte.

El objetivo es minimizar el tiempo total de espera de los buques en el puerto, es decir:

$$\text{mín} \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

La variable de decisión x_{ij} es una variable binaria que indica si el buque i se asigna al atracadero j y c_{ij} es el tiempo de espera del buque i en el atracadero j.

Restricciones:

1. Cada buque debe ser asignado a un sólo atracadero:

$$\sum_{j=1}^{m} x_{ij} = 1 \quad \forall i \in \{1, 2, ..., n\}$$

2. Capacidad máxima específica en cada atracadero:

$$\sum_{i=1}^{n} x_{ij} \leq C_j \quad \forall j \in \{1, 2, ..., m\}$$

3. Solo un buque puede ocupar un atracadero:

$$\sum_{i=1}^{n} x_{ij} \leq 1 \quad \forall j \in \{1, 2, ..., m\}$$

4. La asignación de buques es binaria:

$$x_{ij} \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, n\}, \forall j \in \{1, 2, \ldots, m\}$$

Una vez resuelto mediante el método simplex, se dieron cuenta de las similitudes que tenía con el Problema del Viajante, y esto dio esperanza a seguir intentando resolverlo, ya que veían cada vez más posible una solución válida.

Dantzig, Fulkerson y Johnson especularon sobre la posibilidad de demostrar la optimalidad partiendo de un recorrido óptimo o cercano mediante la adición de pocas ecuaciones adicionales las cuales denominan cortes. Teniendo la solución final, tendrían que ver cómo llegar a ella, lo cual es más sencillo.

Ya habiendo resuelto problemas a pequeña escala, decidieron resolver un problema de 49 ciudades, para ello apostaron cuántos cortes serían necesarias para resolverlo. Al principio se creía que iban a ser necesarias muchas ecuaciones, ya que las posibildades de rutas eran casi infinitas, pero estos tres matemáticos creían que no eran necesarias observar todas las posibles soluciones para resolver el problema de una forma óptima, ya que sabían que había ciertos "caminos" que no llevaban a soluciones óptimas, por lo que no hacía falta explorar todas ellas. Finalmente, la cantidad necesaria fueron 26 ecuaciones adicionales, una muy pequeña cantidad en comparación con lo que pensaban al principio.

En este punto, no solo se resolvió un TSP de gran tamaño, sino que también demostraron que la complejidad estructural de un problema de optimización combinatoria no era insuperable.

Este gran avance implementó conceptos como el "Branch and Bound", que aplicado al TSP se define como una técnica de optimización que busca encontrar la mejor solución posible a través de la exploración sistemática del espacio de soluciones. Esta técnica divide este espacio de soluciones en subconjuntos más pequeños o "ramas", donde cada rama representa una posible solución parcial. Tras esto, se utiliza una estrategia de "poda" para eliminar ramas que no conducen a una solución óptima o que son peores que una solución ya encontrada.

El algoritmo comienza con una solución inicial y, a través de la exploración de ramas y la aplicación de reglas de poda, busca de manera eficiente la solución óptima o una aproximación cercana a ella (Chaves & Miloca, 2018).

Otros enfoques, como la programación dinámica, se aplicaron al TSP, aunque su complejidad limita su eficacia en problemas de tamaño significativo, este enfoque será estudiado en el punto 1.2 mediante la comparación de problemas P vs NP.

Tras ser estudiado por una gran cantidad de matemáticos, se producen grandes avances en las décadas de 1970 y 1980, con la resolución exacta de casos con hasta 2.392 ciudades, utilizando métodos de planos de corte y ramificación. En este punto, ya no se buscaba resolver casos con el mayor número de ciudades posibles, sino cómo hacerlo de la forma más sencilla y tomando el menor tiempo posible.

En la década de 1990, el programa Concorde, fue desarrollado por William Cook, William H. Cunningham, y otros, para resolver el Problema del Viajante (Applegate et al., 2003).

Concorde implementa una variedad de algoritmos avanzados para encontrar soluciones óptimas o muy cercanas a óptimas para instancias grandes del TSP. Utiliza técnicas como la programación lineal entera, optimización combinatoria y métodos heurísticos sofisticados para explorar el espacio de soluciones de manera eficiente (Hahsler & Hornik, 2007).

El TSPLIB, compilado por Reinelt (1992), es una biblioteca de instancias de problemas del Problema del Viajante. Esta biblioteca proporciona una amplia gama de problemas de prueba con diferentes tamaños y características para permitir la comparación de algoritmos y la evaluación del rendimiento de los métodos de resolución del TSP.

Las instancias en TSPLIB provienen de diversas fuentes y cubren una gran variedad de casos, desde problemas pequeños con unas pocas ciudades hasta problemas muy grandes con miles de ciudades. Los investigadores y profesionales utilizan TSPLIB como referencia estándar para probar y comparar nuevos algoritmos y enfoques para resolver el TSP. Este programa está adecuado para su uso no sólo en el software Concorde, sino en otros más usados como R o Python.

Una vez definidos los términos principales, trataremos de comprender este complejo problema, explorando sus orígenes históricos, formulaciones matemáticas, algoritmos de resolución exactos y heurísticos y aplicaciones prácticas con un ejemplo logístico que será resuelto. A través del análisis de distintos algoritmos examinaremos diferentes estrategias para abordarlo, desde métodos exactos hasta enfoques heurísticos y metaheurísticos.

Tras un primer apartado con los objetivos del trabajo y la compresión del contexto histórico nuestro primer capítulo tratará de la Teoría de grafos y el estudio de la complejidad computacional en problemas P vs NP.

El segundo capítulo se centrará en los distintos métodos de solucion del TSP, concretamente en los métodos exactos.

El siguiente capítulo estudiará la resolución del algoritmo mediante métodos heurísticos.

Nuestro capítulo final tratará de resolver un ejemplo real de un Problema del viajante en su parte logística, más concretamente en la optimización de la ruta de un comerciante a los distintos talleres oficiales de Volvo Trucks de su zona, un ejemplo a pequeña escala que estudiará todas las difentes rutas que puede tomar, de entre aproximadamente 5000, eligiendo la más óptima mediante el algoritmo de resolución escogido.

Tras ser esto resuelto realizaremos un pequeño ejemplo de aplicación web que logrará resolver problemas similares al propuesto.

Por último, se ofrecerá la bibliografía utilizada para la realización de este trabajo. En esta encontraremos diversos libros, artículos y páginas web. Además, anexo, encontraremos el código en R que será utilizado para el desarrollo de nuestra aplicación web.

Objetivos

En este trabajo tendremos los siguientes objetivos:

- Investigar y analizar las bases teóricas así como la formulación matemática del Problema del Viajante.
- Explorar tanto algoritmos exactos como heurísticos para encontrar soluciones óptimas y aproximadas de dicho problema, respectivamente.
- Realizar un análisis exhaustivo de los algoritmos seleccionados, evaluando su rendimiento y complejidad computacional.
- Desarrollar un programa en el entorno de programación R para implementar y ejecutar dichos algoritmos en ejemplos concretos.

Capítulo 1

Consideraciones previas

La teoría de grafos y la complejidad computacional son dos áreas fundamentales en informática que se entrelazan en la resolución de una variedad de problemas prácticos y teóricos. Mientras que la teoría de grafos proporciona herramientas para modelar y analizar relaciones entre entidades, la complejidad computacional estudia los recursos necesarios para resolver problemas computacionales.

1.1. Teoría de grafos

La teoría de grafos, también conocida como teoría de gráficas, es una disciplina matemática cuyo objetivo es estudiar las características y propiedades de las estructuras conocidas como grafos. La teoría de grafos se inició gracias a un problema turístico que resolvió Leonhard Euler, redactado en 1736: El problema de los puentes de Königsberg. Se plantea en Königsberg, ciudad conocida por sus siete puentes que conectan cuatro áreas de tierra separadas por dos ríos. El desafío es encontrar la ruta que cruce cada puente exactamente una vez y regrese al inicio(Alvarez-Nuñez, 2013).

Euler abordó este problema creando un grafo sobre los puentes, donde las tierras son vértices y los puentes son aristas. Demostró que tal ruta no es posible, no puede existir un camino que cruce todas las aristas exactamente una vez. En Königsberg, cada área de tierra tenía un número impar de puentes, lo que impedía encontrar la ruta deseada (Godsil & Royle, 2001).

1.1.1. Definición y propiedades de los grafos

1.1.1.1. Tipos de grafos

Se da una serie de definiciones importantes relativas a los tipos de grafos (Rosen, 2012).

Definición 1.1 *Un grafo (no dirigido) $G = (V, E)$ consiste en un conjunto V de vértices y un conjunto E de pares (no ordenados) de vértices llamados aristas. Cada arista tiene asociado uno o dos vértices, llamados extremos de la arista.*

Definición 1.2 *Un grafo dirigido $G = (V, E)$ consiste en un conjunto de vértices V y un conjunto E de aristas dirigidas. Cada arista dirigida está asociada a un par ordenado de vértices. La arista dirigida asociada al par ordenado (u, v) se dice que empieza en u y termina en v.*

Definición 1.3 *Grafo simple. $G = (V, E)$ se denomina grafo simple si V es un conjunto finito de vértices y E un conjunto de pares no ordenados de vértices distintos a los que denominaremos aristas no dirigidas.*

$$G = (V, E) \qquad E \subseteq \binom{V}{2}$$

Definición 1.4 *Multigrafo. $G = (V, E)$ se denomina multigrafo si $V = \{a_1, \ldots, a_n\}$ es un conjunto finito de vértices y E es una familia finita de pares no ordenados de vértices*

$$G = (V, E) \qquad E \subseteq V \times V$$

1.1.1.2. Conexión de grafos

Definición 1.5 *Camino simple. Un camino en G es un camino simple si no se repiten los vértices por los que pasa.*

Definición 1.6 *Ciclo. Un camino simple en G es un ciclo si es un camino que sale y llega al mismo vértice.*

Definición 1.7 *Recorrido. Un camino en G es un recorrido si no se repiten las aristas por las que pasa*

Definición 1.8 *Circuito. Un camino en G es un circuito si es un recorrido que empieza y termina en el mismo vértice.*

Definición 1.9 *Un grafo completo de n vértices, denotado por K_n, es un grafo simple que contiene exactamente una arista entre cada par de vértices distintos.*

$$G = (V, E) \qquad V = \{v_1, v_2, \ldots, v_n\} \qquad E = \left\{ \left(v_i, v_j\right) | \quad i \neq j,\ 1 \leq i, j \leq n \right\}$$

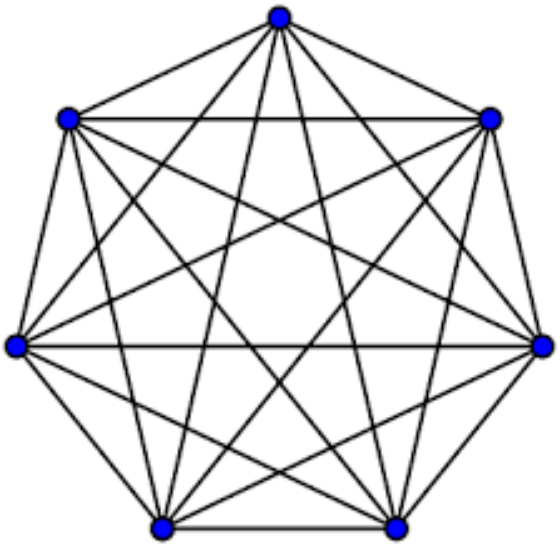

Figura 1.1: Grafo Completo

Definición 1.10 *Grafo conexo. Un grafo no dirigido $G = (V, E)$ se denomina conexo si para cada par de vértices u, v en V, existe al menos un camino que los conecta, es decir, existe una secuencia de aristas $(u, v_1), (v_1, v_2), \ldots, (v_{k-1}, v)$ tal que $\{ v_1, v_2, \ldots, v_{k-1} \}$ son vértices distintos en V.*

1.1.2. Grafos Eulerianos y Hamiltonianos

Dos conceptos importantes en la teoría de grafos son los grafos eulerianos y los grafos hamiltonianos, que representan diferentes propiedades de conectividad y recorrido en los grafos.

1.1.2.1. Grafos Eulerianos

Un grafo euleriano es un grafo en el cual es posible recorrer todas las aristas exactamente una vez y regresar al punto de partida. Este tipo de grafo lleva el nombre de Leonhard Euler, quien en 1736 resolvió el famoso Problema de los Puentes de Königsberg, sentando así las bases de la teoría de grafos (Núñez et al., 2004).

Ejemplo de los puentes de Königsberg

La ciudad de Königsberg está dividida por un río en cuatro áreas distintas. Existen siete puentes que conectan estas áreas. ¿Es factible partir desde cualquier punto, cruzar cada puente exactamente una vez y regresar al lugar de inicio?

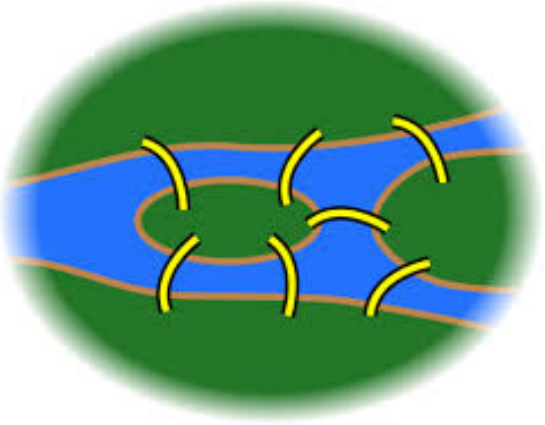

Figura 1.2: Puentes de Königsberg

Euler reformuló el problema al abstractizarlo utilizando la teoría de grafos. Representó las tierras como vértices y los puentes como aristas, simplificando la disposición geográfica de la ciudad en un grafo.

Euler demostró que tal ruta no era posible debido a restricciones topológicas en el grafo resultante. Observó que si un grafo tiene más de dos vértices con un número impar de aristas conectadas a ellos (grado impar), no es posible encontrar un camino que pase exactamente una vez por todas las aristas. En el caso de Königsberg, cada tierra tenía un número impar de puentes que la conectaban, lo que hacía imposible encontrar la ruta deseada.

A partir de esta solución, tenemos el siguiente Teorema de Euler (1741):

En un grafo conexo $G = (V, E)$, donde V es el conjunto de vértices y E es el conjunto de aristas, un grafo es euleriano si y solo si todos sus vértices tienen grado par.

Demostración: El recorrido comienza en el vértice a y sigue hacia un vértice adyacente, al cual llamaremos (a, b). Esta arista incrementa en uno el grado de a. Cada vez que el recorrido pasa por un vértice, se incrementa su grado en dos: uno por la entrada y otro por la salida. Finalmente, el recorrido concluye en el punto de partida, añadiendo uno más al grado de a. Por lo tanto, si un grafo conexo tiene un circuito de Euler, cada vértice debe tener un grado par.

1.1.2.2. Grafos Hamiltonianos

Un grafo hamiltoniano es un grafo en el cual es posible encontrar un ciclo que visite cada vértice exactamente una vez. Estos grafos reciben su nombre en honor a Sir William Rowan Hamilton, un matemático irlandés del siglo XIX.

Juego icosiano

El Juego Icosiano, también conocido como el Juego de los 20 vértices, es un rompecabezas geométrico que consiste en encontrar un ciclo hamiltoniano en un dodecaedro regular tridimensional, es decir, un ciclo que visite cada vértice exactamente una vez.

El juego se presenta como una red de 20 vértices (esquinas) y 30 aristas (conexiones entre las esquinas) que forman las 12 caras del dodecaedro. El problema consistía en hallar un circuito que pasara exactamente una vez por cada vértice retornando al vértice de partida, lo que hoy se conoce como circuito hamiltoniano(Hamilton, 1858).

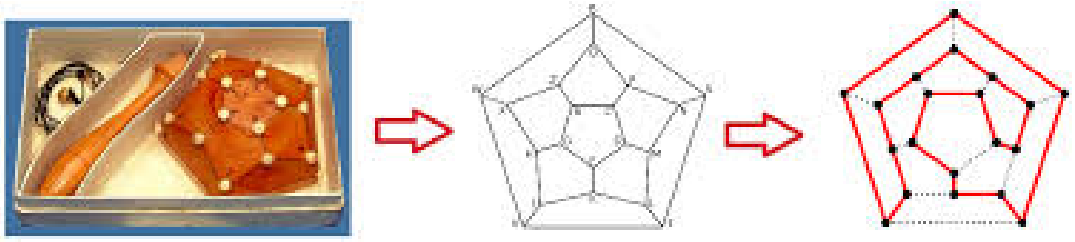

Figura 1.3: Juego Icosiano

Dado un grafo (V, E) existe un camino o circuito Hamiltoniano si y sólo si se cumplen ciertas condiciones. Dos de los teoremas más importantes son:

Teorema 1.1 *Teorema de Dirac (1952). Sea $G = (V, E)$ un grafo simple con n vértices donde $n \geq 3$. Si el grado de cada vértice en G es al menos $\frac{n}{2}$, entonces G contiene un ciclo hamiltoniano.*

Teorema 1.2 *Teorema de Ore (1960). Sea $G = (V, E)$ un grafo simple con n vértices donde $n \geq 3$. Si para cada par de vértices no adyacentes u y v en G, el grado de u más el grado de v es al menos n, entonces G contiene un ciclo hamiltoniano.*

Estos dos teoremas son condiciones necesarias pero no suficientes para saber si un grafo es Hamiltoniano.

1.1.3. Formulación del Problema del Viajante

Dado un grafo completo ($G = (V, E)$), denominamos V Al conjunto de ciudades y E al conjunto de todas las posibles aristas entre las ciudades. Los pesos asociados serán la distancia entre las ciudades correspondientes. El objetivo es encontrar el ciclo hamiltoniano de peso mínimo (Crehuet-Lucas, 2022), es decir, una ruta que visite cada ciudad exactamente una vez y regrese al punto de partida, minimizando la suma de los pesos de las aristas recorridas.

Matemáticamente, el problema se puede formular como un problema de optimización con la siguiente función objetivo y restricciones:

Función Objetivo:

$$\text{mín} \sum_{(i,j)\in E} c_{ij} x_{ij}$$

Donde c_{ij} es la distancia asociada a la arista entre dos ciudades i y j, y x_{ij} es una variable binaria que indica si la arista entre las dos ciudades está incluida en la solución

Restricciones:

1. Cada ciudad debe ser visitada exactamente una vez:

$$\sum_{j\neq i} x_{ij} = 1 \quad \forall i \in V$$

2. No se pueden hacer subciclos:

$$\sum_{(i,j)\in S} x_{ij} \leq |S| - 1 \quad \forall S \subset V, |S| \geq 2$$

3. Las variables de decisión son binarias:

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in E$$

1.2. Teoría de la complejidad computacional

En la teoría de la complejidad computacional, los problemas se clasifican en diferentes categorías según el tiempo y los recursos que tenemos para resolverlos. Una de las principales clasificaciones es la de los problemas de clase P y los problemas de clase NP.

Los problemas de clase P son aquellos que pueden resolverse en tiempo polinomial, es decir, el tiempo necesario para resolverlos crece de manera polinómica en función del tamaño de la entrada del problema. Por otro lado, los problemas de clase NP son aquellos cuyas soluciones pueden verificarse en tiempo polinomial, pero no necesariamente pueden resolverse en tiempo polinomial.

En esta sección veremos que el Problema del Viajante se clasifica como un problema NP-Completo.

INTRODUCCIÓN

Definición 1.11 *Un algoritmo es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema. Esto quiere decir que un algoritmo nos indica el modo de resolver un problema planteado mediante una serie de intrucciones o pasos que nos llevan a la solución que buscamos. Es importante tener en cuenta que un algoritmo debe tener un número finito de pasos(Crehuet-Lucas, 2022).*

En los siguientes párrafos se dan una serie de definiciones importantes relativas a los algoritmos (Domínguez-Pérez, 2020).

Definición 1.12 *Problema computable: Aquel para el que existe un algoritmo finito en espacio y tiempo capaz de resolverlo.*

Definición 1.13 *Complejidad algorítmica: Clasificación de los algoritmos según la cantidad de tiempo y espacio que "ocupan".*

Definición 1.14 *Complejidad computacional: Clasificación de los problemas según el algoritmo que los resuelve.*

Cuando hablamos del algoritmo que resulve un problema siempre nos referimos al algoritmo más eficiente. La eficiencia de un algoritmo se mide en términos de los recursos que consume, como el tiempo, el espacio (memoria) y la capacidad de procesamiento. Es deseable que un algoritmo utilice menos recursos para llevar a cabo su tarea.

1.2.1. Máquina de Turing

Uno de los enfoques más destacados para definir un algoritmo es el concepto de máquina de Turing, propuesto por (Turing, 2004).

La máquina de Turing es una quíntupla que se define como:

$$M = (Q, \Sigma, \Gamma, \delta, q_0)$$

Donde:

Q es el conjunto finito de estados.

Σ es el conjunto finito de símbolos de entrada.

Γ es el conjunto finito de símbolos de la cinta.

δ es la función de transición.

q_0 es el estado inicial.

Introduciendo el tema de la complejidad computacional y para poder explicar más tarde los problemas P vs NP es necesario definir los siguientes términos (Pérez-Jiménez, 2019):

Máquina de Turing Determinista

Una máquina de Turing determinista es un modelo teórico de computación que consiste en una cinta infinita dividida en celdas, una cabeza lectora/escritora que puede desplazarse a lo largo de la cinta, y un conjunto finito de estados y reglas de transición. En cada paso, la máquina lee el símbolo presente en la cinta, cambia a un nuevo estado según su tabla de transición y escribe un símbolo en la cinta. Las reglas de transición son deterministas, lo que significa que para cada combinación de estado actual y símbolo leído, existe una única acción que la máquina puede realizar.

Máquina de Turing no Determinista

Una máquina de Turing no determinista es similar a una máquina de Turing determinista, pero en este caso en cada paso la máquina puede tomar más de una decisión, cada una siguiendo una

regla de transición diferente. La decisión de qué camino tomar puede depender de una decisión no determinista dependiendo del problema que tengamos, el algoritmo elegirá qué hacer.

1.2.2. Problemas P vs NP

Cuando medimos la eficacia computacional de un problema, existes dos grandes grupos que podemos definir (Cabezas, 2014) como:

Problemas de clase P: Los problemas de la clase P son aquellos que pueden resolverse en tiempo polinomial por una máquina de Turing determinista. En otras palabras, existe un algoritmo eficiente que puede encontrar una solución al problema en tiempo polinomial en función del tamaño de la entrada.

Un problema es de clase P si existe un algoritmo A que resuelve el problema en un tiempo $O\left(n^k\right)\big)$, donde n es el tamaño de la entrada y k es una constante.

Problemas de clase NP: Los problemas de la clase NP son aquellos cuyas soluciones pueden verificarse en tiempo polinomial por una máquina de Turing no determinista. Esto significa que si se presenta una posible solución al problema, se puede verificar su validez en tiempo polinomial.

Un problema es de clase NP si dada una solución propuesta c, el tiempo que tarda una máquina de Turing determinista para verificar si c es una solución es $O\left(n^k\right)$.

1.2.2.1. NP-Completo

En teoría de la complejidad computacional, dentro de los problemas NP, nos encontramos con la clase de problemas NP-completos, los cuales tienen una propiedad particular: cualquier problema en NP puede ser reducido, de manera eficiente, a cualquiera de los problemas NP-completos. Se considera que los problemas NP-completos son los más difíciles dentro de la clase NP(Garey, 1979).

Tienen una gran importancia en la complejidad computacional, ya que está demostrado que si se encontrara una solución polinómica para un problema NP-completo, todos los problemas de NP podrían ser resueltos en tiempo polinómico.

Si un problema NP-completo, denominado A, se demostrara que no puede ser resuelto en tiempo polinómico, entonces ninguno de los problemas NP-completos podría ser resuelto en tiempo polinómico. Esto se debe a que si otro problema NP-completo diferente de A pudiera ser resuelto en tiempo polinómico, entonces A también podría reducirse a él, según la definición de NP-completos, y ser resuelto en tiempo polinómico. Sin embargo, es importante destacar que pueden existir problemas en NP que no sean NP-completos y que, no obstante, tengan una solución en tiempo polinómico, aunque el resto no la tenga.

¿Por qué el Problema del Viajante es un problema NP-Completo?

El Problema del Viajante es un problema NP-completo debido a su naturaleza combinatoria y su capacidad para expresar cualquier problema en NP mediante una reducción polinómica (Asociación Nacional de Estudiantes de Matemáticas, 2018).

Para demostrar que el TSP es NP-completo, se puede utilizar un problema como el Problema del Ciclo Hamiltoniano, que ya se ha demostrado que es NP-completo y transformarlo al Problema del Viajante.

El Problema del Ciclo Hamiltoniano es un problema clásico en teoría de grafos que consiste en determinar si un grafo contiene un ciclo hamiltoniano, es decir, un ciclo simple que pasa por cada vértice exactamente una vez.

Formalmente, dada un grafo no dirigido $G = (V, E)$, donde V es el conjunto de vértices y E es el conjunto de aristas, el Problema del Ciclo Hamiltoniano busca determinar si existe un ciclo hamiltoniano en G. Es decir, se busca un camino que visite todos los vértices del grafo una vez y regrese al vértice de inicio. Es sencillo relacionar este problema con el TSP, ya que busca el mismo ciclo.

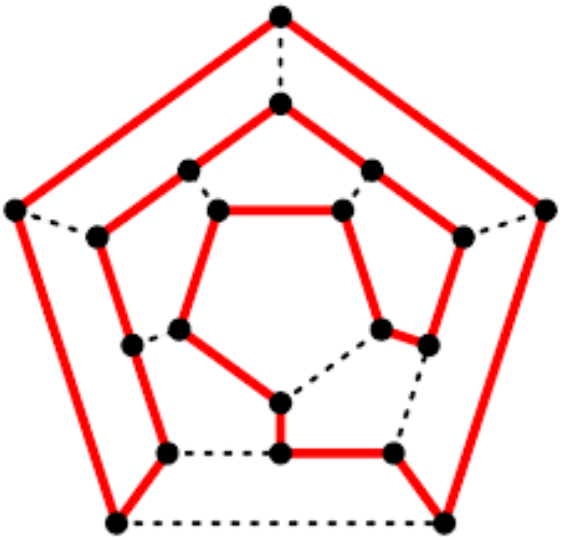

Figura 1.4: Camino Hamiltoniano

Dado que la transformación es eficiente y que el Problema del Ciclo Hamiltoniano ya es NP-completo, se demuestra que el TSP también es NP-completo.

A pesar de que el TSP es un problema NP-completo, ya que aún no se ha encontrado la manera de obtener una solución exacta en un tiempo razonable, se han descubierto distintas técnicas para la resolución de este problema, como estudiaremos en el siguiente capítulo.

Capítulo 2

Métodos de solución con algoritmos exactos

Introducción

Los algoritmos exactos son algoritmos que garantizan encontrar la solución óptima para un problema dado (Golden et al., 1980). Estos algoritmos exploran todas las posibles soluciones en busca de la mejor solución posible. A diferencia de los algoritmos aproximados, que pueden proporcionar soluciones cercanas a la óptima pero no garantizan encontrarla, los algoritmos exactos son capaces de encontrar la solución óptima si se les da suficiente tiempo y recursos computacionales.

Aunque estos métodos garantizan la solución óptima, su complejidad crece exponencialmente con el número de ciudades, lo que los hace inviables para instancias grandes. A medida que el número de ciudades crece, el espacio de soluciones posibles aumenta exponencialmente, haciendo que los métodos exactos se vuelvan ineficientes en términos de tiempo y recursos computacionales.

Los métodos exactos se aplican principalmente a instancias pequeñas y sirven como referencia para evaluar la eficacia de otros métodos.

A continuación se estudian tres ejemplos de algoritmos exactos con los que poder resolver el Problema del Viajante.

2.1. Algoritmo de Fuerza bruta

Los algoritmos de fuerza bruta intentan resolver un problema generando todas las posibles soluciones y verificando una por una si cumplen con las condiciones necesarias para ser la respuesta correcta, es decir, explora cada una de las posibilidades. Si la solución existe, con el algoritmo de fuerza bruta es imposible no encontrarla.

Para el caso del problema estudiado, este algoritmo consiste en enumerar todas las posibles permutaciones de las ciudades y calcular la longitud del recorrido para cada una de ellas, es decir, generar todas las posibles rutas a partir de las n ciudades dadas. Como nuestro problema es de minimización, se selecciona el recorrido con la longitud mínima como la solución óptima.

A pesar de que es un algoritmo óptimo, pues siempre encuentra solución si esta existe, su complejidad factorial hace que sea imposible computacionalmente de resolver por tiempo y espacio para un número de ciudades significativo. Aunque este algoritmo puede ser útil y sencillo para un número pequeño de datos(Tello, 2018).

Para lograr entender este algoritmo, se resuelve el siguiente problema:

Ejemplo

Dado un conjunto de ciudades en España, cada una con coordenadas geográficas y distancias entre ellas, se busca encontrar el camino más corto que pase por todas las capitales de España y regrese al punto de partida, minimizando la distancia total recorrida.

Solución

Sea $n = 52$ el número de capitales en España y d_{ij} la distancia entre la ciudad i y la ciudad j, donde $1 \leq i, j \leq 52$. Entonces, el algoritmo de fuerza bruta para el TSP se puede expresar de la siguiente manera:

1. Generar todas las permutaciones posibles de las $n = 52$ capitales.
2. Para cada permutación, calcular la longitud del recorrido total sumando las distancias entre ciudades consecutivas en la permutación.
3. Seleccionar el recorrido con la longitud mínima como la solución óptima.

Las permutaciones posibles son $52! = 8{,}0658175 \cdot 10^{67}$, y entre estas permutaciones se encuentra la solución óptima, a pesar de ello, como se puede comprobar, computacionalmente es muy costoso resolver este problema con el algoritmo de fuerza bruta, por lo que conviene estudiar otros algoritmos exactos o heurísticos.

2.2. Algoritmo de ramificación y acotación

El algoritmo de ramificación y acotación es una técnica utilizada para resolver problemas de optimización combinatoria. Al igual que el algoritmo de fuerza bruta, funciona explorando todas las soluciones posibles, pero lo que caracteriza a este algoritmo es la capacidad de elegir explorar sólo las soluciones que tienen un resultado óptimo, eliminando las que no van por buen camino; a este proceso se le denomina “Poda”. No sólo se elimina una solución en cada poda, si no todas las ramas que seguirían a esa posibilidad.

El paso a paso para resolver el algoritmo es el siguiente:

- Convertir el problema de programación en un problema relajado, esto es, eliminando la restricción de números enteros (Relajación Lagrangiana).
- Resolver el problema relajado.
- Si la solución a este no cumple la restricción de números enteros, se imponen dos nuevas restricciones sumando y restando uno a nuestra solución para asegurarnos que se elimina la región de solución que no lleva a un número entero
- Repetimos este paso hasta encontrar soluciones enteras. A estos pasos se les denomina “Ramificación”, y como su nombre indica, de cada solución no entera sacamos dos soluciones.
- Una vez tenemos los candidatos a solución óptima, se aplica la técnica de “Acotación”. En este paso vamos a decidir si seguir ramificando o no, esto va a depender de los límites inferior y superior que fijaremos previamente.
    - Si se decide seguir ramificando, se vuelve a repetir el proceso.
    - Si se decide no seguir ramificando, se elimina ese nodo, lo que llamamos “acotación”.

En este algoritmo, se comienza con un nodo raíz que representa el estado inicial del problema. Luego, se generan sucesivamente nodos hijos a partir del nodo actual. Cada nodo generado se evalúa para determinar si podría conducir a una solución óptima o no. Si se determina que un nodo

no puede mejorar la solución actual, se descarta o "poda" para evitar explorar sus nodos hijos, ya que estos tampoco llevan a ninguna solución óptima.

Un concepto importante para entender este tipo de algoritmo es el uso de límites (o cotas) inferiores y superiores. Son valores que ayudan a guiar el proceso de toma de decisiones y se utilizan para identificar y descartar ramas que no llevan a la solución óptima.

El **límite inferior** representa el costo mínimo posible para una solución. Si el límite inferior de un subproblema es mayor o igual que el costo de una solución conocida, esa rama puede ser descartada porque no ofrecerá una mejor solución. El límite inferior puede calcularse sumando los costos más bajos posibles para cada nodo o ciudad.

Por otro lado, el **límite superior** es el costo máximo aceptable para una solución. Generalmente se establece con base en una solución conocida o inicial y puede ajustarse a medida que se encuentran soluciones mejores

Los valores de estos límites se fijan habiendo estudiado previamente el problema y con ayuda de otros algoritmos exactos o heurísticos.

Este tipo de algoritmo resulta muy útil para resolver problemas como el TSP con muchas posibles soluciones, ya que ayuda a agilizar el proceso de búsqueda de una solución óptima.

Es posible plantear el Problema del Viajante como un problema de programación lineal minimizando la siguiente función:

$$\sum_{i,j=1}^{n} d_{i,j} x_{i,j}$$

Sujeta a las siguientes restricciones:

$$\sum_{j=1}^{n} x_{ij} = 1$$

$$\sum_{i=1}^{n} x_{ij} = 1$$

$$u_i - u_j + n x_{ij} \leq n - 1$$

$$x_{ij}, u_i, u_j \geq 0,$$
$$i = 1, \ldots, n,\ j = 1, \ldots, n,$$

para todo $1 \leq i \neq j \leq n$.

Como se puede comprobar, las restricciones esta vez son menos estrictas, a este proceso se le denomina **Problema Relajado**.

El algoritmo de ramificación y poda (Branch and Bound) ha demostrado ser más adecuado que el algoritmo de fuerza bruta para resolver el Problema del Viajante por ser más eficiente en el tiempo que necesita para resolverlo. En lugar de evaluar todas las posibles soluciones, el algoritmo de ramificación y poda se enfoca sólo en las permutaciones que tienen probabilidades de llevar a la solución óptima.

2.2.1. Técnicas de relajación: Relajación Lagrangiana

En el último algoritmo estudiado hemos introducido el concepto de Problema Relajado:

Definición 2.1 *Dado un problema lineal entero, se llama problema relajado al mismo modelo lineal pero prescindiendo de la restricción de que las variables sean enteras.*

El método de relajación Lagrangiana permite simplificar problemas complejos como el TSP al relajar restricciones y buscar cotas inferiores mediante el uso de multiplicadores de Lagrange, facilitando la solución de problemas de programación entera y combinatoria(Serna, Elena et al., 2021).

Matemáticamente, para un problema de minimización con restricciones de desigualdad, el problema original se puede escribir como:

$$\begin{aligned} \text{mín} \quad & f(x) \\ \text{sujeto a} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m, \end{aligned}$$

donde $f(x)$ es la función objetivo y $g_i(x)$ son las restricciones.

En la relajación Lagrangiana, las restricciones se agregan a la función objetivo con multiplicadores de Lagrange $\lambda_i \geq 0$, y el problema resultante es:

$$\begin{aligned} &\text{máx} \inf_x L(x, \lambda) \\ &L(x, \lambda) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x), \end{aligned}$$

donde el valor de $\inf_x L(x, \lambda)$ proporciona una cota inferior para el problema original.

Aplicación al Problema del Viajante: La restricción que buscamos relajar mediante los multiplicadores de Lagrange con el método de acotación y ramificación es la de la presencia de subcircuitos. En esta formulación, la restricción de "No subcircuitos" puede ser complicada de manejar. Al aplicar la relajación Lagrangiana, se puede relajar esta restricción y agregarla a la función objetivo con otro multiplicador de Lagrange, y el problema relajado se convierte en:

$$\begin{aligned} &\text{máx} \inf_x L(x, \lambda) \\ &L(x, \lambda) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} + \lambda \cdot (\text{Subcircuitos}), \end{aligned}$$

Donde λ es el multiplicador de Lagrange que penaliza la presencia de subcircuitos.

2.3. Programación dinámica: Algoritmo de Held-Karp

El algoritmo de Held-Karp (Alfaro, 2020), propuesto a principios de los años sesenta, es un método de programación dinámica para encontrar la solución óptima del Problema del Viajante. Este algoritmo se basa en la idea de que cualquier camino parcial de un circuito óptimo también debe ser óptimo para los subproblemas que lo componen.

La idea del algoritmo es, primero, encontrar para cada ciudad i (excluyendo la ciudad 1, que es la de inicio y fin), el coste mínimo de los caminos que empiezan en la ciudad 1 y terminan en la ciudad i, visitando todas las demás ciudades exactamente una vez. Este coste se denota como $\text{cost}(i)$. A continuación, encontrar el coste total del ciclo para cada ciudad i, que se obtiene sumando el $\text{cost}(i)$ y la distancia desde la ciudad i de regreso a la ciudad 1, denotada como $d(i, 1)$. Y por último, el coste del ciclo óptimo será el mínimo de estos valores para cada i.

Una explicación detallada de este algoritmo sería la siguiente:

Se define $C(S, i)$ como el coste mínimo del camino que termina en la ciudad i y ha visitado todas las ciudades en el subconjunto S (en el que no se incluye la ciudad 1 pero sí la ciudad i).

Se generan todos los subconjuntos posibles de ciudades, excluyendo la ciudad 1 e incluyendo la ciudad i. Estos subconjuntos van desde el conjunto $\{i\}$ hasta el conjunto completo de ciudades (excluyendo la ciudad 1).

Inicialmente, para cada ciudad i, el coste de viajar directamente desde la ciudad 1 es simplemente la distancia entre ellas:

$$C(\{i\}, i) = d(1, i)$$

Para cada subconjunto S de tamaño mayor a uno que no incluya la ciudad 1 e incluya la ciudad i, $C(S, i)$ se calcula como:

$$C(S, i) = \min_{j \in S, j \neq i} \{C(S - \{i\}, j) + d(j, i)\}$$

$C(S - \{i\}, j)$ es el coste de llegar a la ciudad j pasando por todas las ciudades en el subconjunto S menos la ciudad i, y $d(j, i)$ es la distancia entre las ciudades j e i.

El coste mínimo del ciclo completo se obtiene considerando todos los caminos que terminan en cualquier ciudad i (excluyendo la ciudad 1) y regresan a la ciudad 1:

$$C(\{1, 2, \ldots, n\}, 1) = \min_{i \neq 1} \{C(\{2, \ldots, n\}, i) + d(i, 1)\}$$

El algoritmo resuelve problemas con un número pequeño de instancias, ya que se vuelve impráctico para matrices más grandes debido a su complejidad $O\left(n^2 \cdot 2^n\right)$

2.4. Programa Concorde

Concorde (Cook et al., 1994) es un código informático que resuelve el Problema del Viajante así como otros problemas relacionados con la optimización de redes. El código está escrito en el lenguaje de programación *ANSIC* y está disponible para uso académico.

Según los autores de la fuente de información, Concorde es el programa más rápido de resolución de el Problema del Viajante y similares de la actualidad.

El solucionador de Concorde utiliza el método estudiado anteriormente de planos de corte, resolviendo iterativamente las distintas posibilidades de solución de programación lineal del Problema del Viajante.

El tiempo necesario para el solucionador de Concorde para una instancia puede variar considerablemente según el número de ciudades y la complejidad del problema.

Se han realizado varias pruebas con un gran número de ciudades que reiteran la importancia de este paquete. Por ejemplo, en 2005, se resolvió un problema con 33.810 instancias o puntos usando el paquete “Concorde TSP Solver”. Este encontró el menor recorrido comprobado para resolver el problema, un recorrido de entre 66.048.945 posibilidades. Según varias fuentes, este estudio duró casi 16 años.

Como otro ejemplo, en 2006, tenemos que Concorde logró resolver un problema con el mayor número de ciudades resuelto, 85.900.

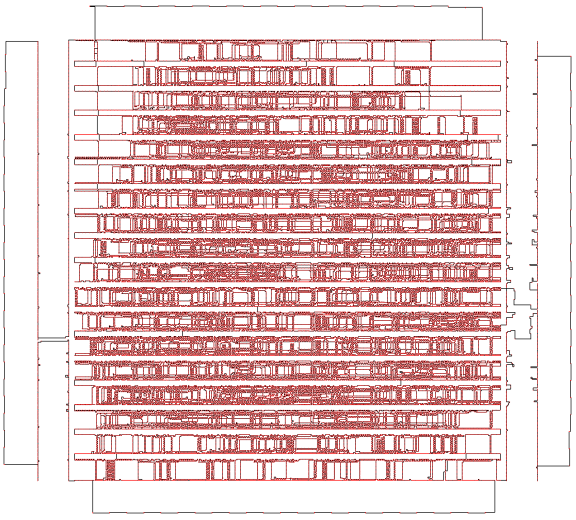

Figura 2.1: Solución óptima de un TSP para 85900 nodos hallada con Concorde (Fuente: Universidad de Cadiz (2021))

Como conclusión, este programa usa algoritmos estudiados como los Planos de Corte y el método de ramificación y acotación para lograr resolver problemas con un gran número de ciudades.

El paquete Concorde se puede usar para fines académicos en R entre otros programas.

Capítulo 3

Métodos de solución con algoritmos heurísticos

Introducción

Como se ha estudiado, el Problema del Viajante es NP-completo, por esta razón los métodos exactos no sirven en muchas ocasiones para instancias grandes. Para esto son usados los algoritmos heurísticos, los cuales encuentran soluciones aproximadas de manera eficiente. Estos métodos no garantizan la solución exacta, pero ofrecen resultados cercanos al óptimo en tiempos razonables.

Al tener la condición de que da resultados aproximados, es importante tener en cuenta la calidad de solución, así como los tiempos empleados.

Los métodos heurísticos son esenciales para resolver el TSP a gran escala, a continuación se estudian algunos de los métodos heurísticos más comunes.

3.1. Algoritmo del vecino más cercano (KNN)

El Algoritmo del vecino más cercano, o K-Nearest-Neighbours, es uno de los primeros algoritmos que fueron estudiados para resolver el Problema del Viajante de forma aproximada (Rosenkrantz et al., 1977), ya que, por lógica, este algoritmo genera rápidamente una camino corto, pero muchas veces no el más óptimo. Además de ser usado en espacios métricos y distancias como en el caso del Problema del Viajante, también es utilizado en otras disciplinas como la búsqueda en bases de datos o citas; un ejemplo es buscar palabras similares.

El algoritmo del Vecino más cercano funciona utilizando distintas mediciones de distancias como la distancia Euclidiana o Manhattan, entre otras. En nuestro problema, al partir de una matriz de distancias, usaremos la distancia en kilómetros entre las diferentes ciudades.

El método estudiado comienza seleccionando un nodo o ciudad como punto de partida y luego construye un camino eligiendo el nodo más cercano al último nodo añadido. El proceso continúa hasta que todas las ciudades están incluidas en el recorrido, como una de las restricciones del Problema del Viajante es que se empiece y se termine el recorrido en el mismo nodo, el último paso (una vez se hayan recorrido todos los nodos) es unir el último con el primero seleccionado.

El paso a paso es el siguiente:

- Seleccionar cualquier nodo como el nodo de inicio. Este será el primer nodo del recorrido.
- Se busca el nodo no visitado más cercano al último agregado al recorrido. Este será el siguiente nodo a incluir en el recorrido.
- Se repite el segundo paso hasta que todos los nodos estén en el recorrido.
- Se une el primer y el último nodo para cerrar el circuito.

A pesar de que la lógica lleva a pensar que este algoritmo lleva al recorrido más corto al unir las cuidades más cercanas, puede no ser así, no garantiza una solución óptima. Esto puede suceder si la ciudad más cercana no necesariamente lleva al recorrido más corto.

En los algoritmos aproximados es importante tener en cuenta no sólo la calidad de la solución sino el número de cálculos. Este dato nos da una idea de la complejidad del algoritmo. El algoritmo del Vecino más cercano tiene una complejidad de $O\left(n^2\right)$, donde n es el número de ciudades o nodos, lo que significa que el número de cálculos necesarios no crece más rápido que n^2. Esto implica que el tiempo de ejecución aumenta cuadráticamente con el número de ciudades, lo que lo hace adecuado para instancias de tamaño pequeño a moderado.

En la búsqueda de una mejor calidad en la solución, el algoritmo puede probar a iniciar desde cada nodo y determinar cuál proporciona el mejor resultado. Esto puede ayudar a encontrar un recorrido más corto y por lo tanto una solución más óptima, pero requiere más cálculos, ya que el algoritmo debe ejecutarse desde cada nodo como punto de partida para comparar los resultados.

3.2. Colonia de hormigas (ACO)

El algoritmo de optimización de colonia de hormigas, o Ant Colony Optimization, se basa en el comportamiento de hormigas reales. En la naturaleza, las hormigas utilizan rastros de feromonas para encontrar caminos cortos entre sus nidos y fuentes de alimento. Las hormigas depositan feromonas mientras se mueven, y la concentración de feromonas influye en las decisiones de otras hormigas, esto permite a la colonia encontrar rutas óptimas (Dorigo & Gambardella, 1997).

Relacionándolo con el Problema del Viajante, el comportamiento de las hormigas se basa en encontrar puntos (ciudades) cercanas y llegar a ellas con el menor costo.

Para la solución, este algoritmo combina la distancia entre ciudades y el nivel de feromonas.

El paso a paso sería el siguiente:

- En primer lugar, se establecen parámetros como el número de hormigas, el factor de evaporación de las feromonas, la influencia de las feromonas y la influencia de la heurística.
- Cada hormiga comienza en una ciudad aleatoria. La hormiga elige la siguiente ciudad para visitar basada en la siguiente regla probabilística.

  Definición 3.1 *Dado un grafo completo con ciudades como vértices y distancias como pesos de aristas, la probabilidad de que una hormiga se mueva de la ciudad i a la ciudad j se define como:*

$$p_{ij} = \frac{{\tau_{ij}}^{\alpha} \cdot {\eta_{ij}}^{\beta}}{\sum_{k \in V} {\tau_{ik}}^{\alpha} \cdot {\eta_{ik}}^{\beta}},$$

  donde τ_{ij} es la cantidad de feromonas en la arista entre las ciudades i y j,

  $\eta_{ij} = \dfrac{1}{d_{ij}}$ *es la visibilidad, definida como el inverso de la distancia,*

α y β son parámetros que determinan la influencia de las feromonas y de la visibilidad, respectivamente,

y V es el conjunto de ciudades restantes para visitar.

- Cuando cada hormiga ha recorrido la ruta encontrada, parte de las feromonas de cada arco se evaporan.
- Cada hormiga deposita nuevas feromonas en los arcos que ha recorrido. La cantidad de feromonas depositada en cada arco de la ruta encontrada por la hormiga es inversamente proporcional a su longitud.
- Se repiten los pasos anteriores hasta que la solución no mejora significativa de las encontradas en las últimas iteraciones.
- Al final de las iteraciones, la mejor solución encontrada es tomada como la mejor aproximación al Problema del Viajante.

La principal ventaja de este algoritmo es que puede adaptarse a cambios en tiempo real. La principal desventaja es que debido a la formulación y las operaciones que se tienen que realizar, puede ser computacionalmente muy costoso.

Existe en R un paquete llamado "antcolony" que resuelve este algoritmo tan sólo con introducir las variables necesarias.

3.3. Algoritmos de Inserción: Nearest Insertion, Farthest Insertion

Los algoritmos de Inserción son una heurística simple utilizada para resolver el Problema del Viajante (Pérez-de Vargas-Moreno, 2015). La idea principal es construir la ruta óptima insertando las ciudades en una ruta inicial de manera que se minimice el aumento en la distancia total.

El algoritmo, de forma general, comienza construyendo ciclos que visiten sólo algún vértice, para conocer el espacio, y posteriormente extenderse insertando el resto de vértices. En cada paso se aumenta un vértice hasta tener todos completos, lo que formaría un cilo Hamiltoniano en el Problema del Viajante.

Algoritmo en cada iteración *k*

- Se parte de un ciclo inicial con *k* vértices.
- Seleccionar uno de los restantes vértices dependiendo del método elegido.
- Insertar la ciudad *i* donde menos incremente la longitud del ciclo.
- El algoritmo finaliza cuando se ha formado el ciclo hamiltoniano completo.

Hay varios algoritmos heurísticos de inserción. Para el Problema del Viajante son usados dos principales:

1. Nearest Insertion: Con esta variante, en cada iteración, se selecciona la ciudad no visitada que está más cerca de cualquier ciudad existente en la ruta actual. Luego, se inserta en la posición de la ruta que minimiza el aumento en la distancia total.

2. Farthest Insertion: Contrariamente a Nearest Insertion, con Farthest Insertion, en cada iteración, se selecciona la ciudad no visitada que está más lejos de cualquier ciudad existente en la ruta actual. Luego, se inserta en la posición de la ruta que minimiza el aumento en la distancia total.

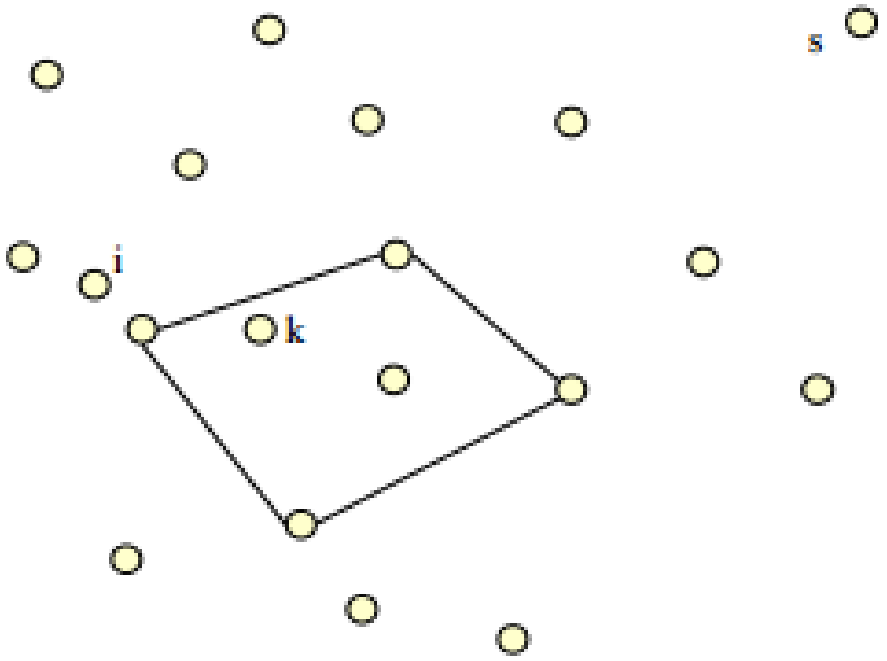

Figura 3.1: Comparativa inserciones

El ciclo que se muestra en la imagen está formado por 4 vértices, es decir $k = 4$, en el siguiente paso del algoritmo hay que determinar el siguiente a insertar. La inserción más cercana escogerá el vértice i, mientras que la inserción más lejana escogerá el vertice s.

El algoritmo de Inserción, con ambas de sus variantes, Nearest Insertion y Farthest Insertion, proporciona una solución heurística eficiente y simple para el Problema del Viajante. Aunque no garantiza la optimalidad de la solución, a menudo produce resultados aceptables en la práctica. La elección entre las variantes depende de las características específicas del problema.

Capítulo 4

Aplicación real del Problema del Viajante

Una vez estudiados de forma teórica los distintos algoritmos tanto exactos como heurísticos para la resolución del Problema del Viajante, se procede a aplicarlos a una pequeña base de datos con un pequeño número de instancias, de este modo podremos realizar un análisis exhaustivo de los algoritmos seleccionados, evaluando su rendimiento y complejidad computacional, uno de los objetivos del trabajo.

4.1. Base de datos

La base de datos escogida se basa en las distancias en kilómetros de distintos concesionarios de camiones Volvo, en concreto, los talleres a 200 km a la redonda de la sede de Talleres Volvo en Madrid.

Los datos han sido recogidos mediante la aplicación Dealer Locator de Volvo Trucks,

(https://www.volvotrucks.es/es-es/tools/dealer-locator.html).

Volvo Dealer Truck Locator es una aplicación que permite a los usuarios localizar concesionarios de camiones Volvo en todo el mundo. Facilita la búsqueda de ubicaciones cercanas, proporcionando detalles como direcciones, contactos y servicios ofrecidos por cada concesionario.

En la siguiente imagen, se ve lo que muestra por pantalla la aplicación cuando partimos de Madrid con un alcance de 200 km. Se puede modificar tanto el punto de partida como el alcance, pudiendo obtener resultados hasta en toda Europa. Además esta aplicación nos da la información de la distancia en kilómetros, por lo que es muy interesante para el estudio de un problema del tipo del Problema del Viajante.

Figura 4.1: Aplicación Volvo Dealer

Tras elegir los 7 concesionarios que usaré para el estudio, el siguiente paso es la creación de la base de datos. La base de datos creada es una matriz de distancias de 8x8.

Definición 4.1 *En matemáticas, ciencias de la computación y teoría de grafos, una matriz de distancias es una matriz cuadrada cuyos elementos representan las distancias entre los puntos, tomados por pares, de un conjunto.*

La aplicación usada para la creación de la matriz de distancias es Excel. A continuación se muestra la matriz de distancias en kilómetros:

	Madrid	Volvo Torrejón	Volvo Valdemoro	Talleres Sanfer	TEC-VIR	Vicente Alvarez Valladolid	Volvo Manzanares	Vicente Alvarez Palencia
Madrid	0	23	28	116	142	154	158	183
Volvo Torrejón	23	0	51	158	163	219	195	245
Volvo Valdemoro	28	51	0	139	191	224	146	281
Talleres Sanfe	116	158	139	0	299	230	220	285
TEC-VIR	142	163	191	299	0	89	335	88
Vicente Alvarez Valladolid	154	219	224	230	89	0	368	47
Volvo Manzanares	158	195	146	220	335	368	0	419
Vicente Alvarez Palencia	183	245	281	285	88	47	419	0

Figura 4.2: Matriz de distancias

Como se puede ver, los 7 concesionarios elegidos son:

1. VOLVO TRUCK CENTER, S.L.U. (Torrejón de Ardoz (Madrid))
2. VOLVO TRUCK CENTER, S.L.U. (Valdemoro (Madrid))
3. TALLERES SANFER, S.A. (Talavera de la Reina (Toledo))
4. TÉCNICAS VEHÍCULOS IND. DE LA RIBERA (Aranda de Duero (Burgos))
5. VICENTE ALVAREZ, S.L. (La Cistérniga (Valladolid))
6. VOLVO TRUCK CENTER, S.L.U. MANZANARES (Ciudad Real)
7. VICENTE ÁLVAREZ (Villamuriel de Cerrato (Palencia))

4.1.1. Enunciado del problema

Un técnico experto necesita visitar todos los concesionarios Volvo en un radio de 200 km desde Madrid. El técnico quiere iniciar su recorrido desde Madrid, visitar cada concesionario exactamente una vez y regresar al punto de partida al finalizar su viaje. El objetivo es encontrar la ruta más óptima que minimice la distancia total recorrida entre los diferentes concesionarios. ¿Cuál es el mejor recorrido para el técnico que cumpla con estas condiciones?

En este caso específico, el Problema del Viajante tiene las siguientes características:

Ítems: Hay 7 concesionarios Volvo dentro de un radio de 200 km desde Madrid.

Punto de Partida y Retorno: El técnico comienza su recorrido en Madrid y debe volver a Madrid al final del recorrido.

Recorrido Único: El técnico debe visitar cada concesionario exactamente una vez durante el recorrido.

Optimización de Distancia: El objetivo es encontrar el recorrido más corto posible que cumpla con las condiciones anteriores.

Restricciones del Problema: El técnico debe visitar cada concesionario una sola vez, sin importar el orden, y al final del recorrido debe regresar a Madrid. Además, el recorrido debe ser el más corto posible en términos de distancia total.

Utilizando la programación entera, tenemos los siguientes puntos:

Definimos variables binarias x_{ij} para cada par de concesionarios i y j (consideramos Madrid como el concesionario 1). Si el recorrido incluye el tramo entre i y j, $x_{ij} = 1$; de lo contrario, $x_{ij} = 0$.

Función Objetivo

La función objetivo es minimizar la distancia total:

$$\text{mín} \sum_{i=1}^{8} \sum_{j=1}^{8} c_{ij} \cdot x_{ij},$$

donde c_{ij} es la distancia entre el concesionario i y el concesionario j.

Restricciones

1. Cada concesionario debe tener exactamente una salida:

$$\sum_{j=1, j\neq i}^{8} x_{ij} = 1, \quad \forall i \in \{1, \ldots, 8\}.$$

2. Cada concesionario debe tener exactamente una entrada:

$$\sum_{i=1, i\neq j}^{8} x_{ij} = 1, \quad \forall j \in \{1, \ldots, 8\}.$$

3. Para evitar subcircuitos, utilizamos las restricciones de Miller- Tucker-Zemlin con variables auxiliares u_i:

$$u_i - u_j + 8 \cdot x_{ij} \leq 7, \quad \forall i, j \in \{1, \ldots, 8\}, i \neq j.$$

4.2. Resolución con algoritmos exactos

En esta sección, se presentará la resolución del problema del comerciante de Volvo planteado anteriormente utilizando diversos algoritmos exactos implementados en R Studio. A continuación se muestran los algoritmos de resolución exactos que se han usado y las soluciones. Dentro de las soluciones se calcula la ruta óptima y el coste, que en este caso es la cantidad de kilómetros que debe recorrer el comerciante. Como se trata de comparar los distintos algoritmos hemos implementado una función en R: "system.time" que calcula el tiempo, y por lo tanto la complejidad computacional que tarda cada algoritmo en resolver el problema.

4.2.1. Algoritmo de Fuerza Bruta

Este enfoque explora todas las posibles permutaciones de los concesionarios Volvo para determinar la ruta de menor coste.

A continuación se muestra el pseudocódigo que resuelve el problema.

```
Inicializar matriz dist_matrix con los valores dados
n <- número de filas de dist_matrix

Definir función calc_distance(ruta, dist_matrix):
    Inicializar total_distance a 0
    Para i desde 1 hasta longitud(ruta) - 1 hacer:
        total_distance <- total_distance + dist_matrix[ruta[i], ruta[i + 1]]
    Fin Para
    Retornar total_distance
Fin Definir

cities <- [2, ... , n]
Generar todas las permutaciones de cities y asignarlas a permutations

# Evaluar todas las rutas y encontrar la de menor distancia
Inicializar min_distance a Inf
Inicializar best_route a NULL

Para i desde 1 hasta número de filas de permutations hacer:
    # Ruta completa empezando y terminando en la ciudad 1
    route <- [1] concatenado con permutations[i, ] concatenado con [1]
    current_distance <- calc_distance(route, dist_matrix)
    Si current_distance < min_distance entonces:
        min_distance <- current_distance
        best_route <- route
    Fin Si
Fin Para

Imprimir "Ruta óptima (Fuerza Bruta):", best_route
Imprimir "Longitud del recorrido (Fuerza Bruta):", min_distance
```

Explicación del código: Se inicializa una matriz de distancias y se define una función para calcular la distancia total de una ruta. Luego, se generan todas las permutaciones posibles de las ciudades (excepto la ciudad inicial) y se evalúa cada ruta posible que comience y termine en la ciudad 1. Para cada ruta, se calcula la distancia total y se la compara con la distancia mínima registrada,

actualizando la mejor ruta y la distancia mínima si se encuentra una ruta más corta. Finalmente, se imprime la ruta óptima y su longitud.

Los resultados obtenidos son los siguientes:

Ruta óptima (Fuerza Bruta): 1 2 5 8 6 4 7 3 1

Longitud del recorrido (Fuerza Bruta): 945

Tiempo: 30.56 segundos

4.2.2. Algoritmo de Ramificación y Acotación (Branch and Bound)

Este método utiliza un enfoque de árbol de decisión, donde se expande y evalúa cada ruta posible utilizando límites inferiores para descartar subárboles completos que no pueden contener la solución óptima.

El pseudocódigo es el siguiente:

```
Inicializar matriz dist_matrix con los valores dados

Definir función branch_and_bound_tsp(dist_matrix):
    n <- número de filas de dist_matrix
    Inicializar best_cost a Infinito
    Inicializar best_path a NULL

    # Cola de prioridades para almacenar los nodos a expandir
    Inicializar pq a lista vacía
    Añadir a pq el nodo con path = [1],
                            cost = 0,
                            visited = [TRUE, FALSE, ..., FALSE]

    Mientras longitud(pq) > 0 hacer:
        # Extraer el nodo con menor coste
        current <- primer elemento de pq
        Eliminar el primer elemento de pq

        current_path <- current$path
        current_cost <- current$cost
        visited <- current$visited

        # Si hemos visitado todas las ciudades, verificar el coste total
        Si longitud(current_path) = n entonces:
            total_cost <- current_cost +
              dist_matrix[current_path[último elemento de current_path],
                          current_path[1]]
            Si total_cost < best_cost entonces:
                best_cost <- total_cost
                best_path <- current_path concatenado con current_path[1]
            Fin Si
        Si no:
            # Expandir el nodo actual
            last_city <- último elemento de current_path
            Para i desde 1 hasta n hacer:
```

```
                Si no visited[i] entonces:
                    new_path <- current_path concatenado con i
                    new_cost <- current_cost + dist_matrix[last_city, i]
                    new_visited <- copia de visited
                    new_visited[i] <- TRUE

                    Si new_cost < best_cost entonces:
                        Añadir a pq el nodo con path = new_path,
                                                cost = new_cost,
                                                visited = new_visited
                    Fin Si
                Fin Si
            Fin Para
        Fin Si

        # Ordenar la cola de prioridades por coste
        costs <- mapear pq para obtener los valores de cost
        pq <- ordenar pq según costs
    Fin Mientras

    Retornar lista con cost = best_cost, path = best_path
Fin Definir

Imprimir "Ruta óptima (Ramificación y acotación):", path
Imprimir "Longitud del recorrido (Ramificación y acotación):", cost
```

Explicación del código: Se inicializa una matriz de distancias y una cola de prioridades para manejar los nodos a expandir. Se comienza con la ciudad 1 y se expanden iterativamente los nodos, calculando rutas parciales y sus costes. Si se visitan todas las ciudades, se verifica y actualiza el coste total de la ruta óptima. Los nodos se expanden agregando ciudades no visitadas a la ruta actual, actualizando el coste y las ciudades visitadas. La cola de prioridades se ordena según los costes para garantizar la expansión del nodo más prometedor. Al finalizar, se retorna y se muestra la mejor ruta y su coste.

Los resultados obtenidos son los siguientes:

Ruta óptima (Branch and Bround): 1 2 5 8 6 4 7 3 1

Longitud del recorrido (Branch and bround): 945

Tiempo: 140.33 segundos

4.2.3. Algoritmo de Held-Karp

Este algoritmo se basa en la reducción del problema a sub-rutas y la construcción de una tabla de costes mínimos para cada subconjunto de ciudades.

El pseudocódigo en R utilizando programación dinámica es el siguiente:

```
Inicializar matriz dist_matrix con los valores dados

Definir función held_karp(dist_matrix):
    n <- número de filas de dist_matrix
```

```
# Inicializar la tabla de costes y rutas
Inicializar C a matriz de Infinitos de tamaño 2^n x n
Inicializar R a matriz de ceros de tamaño 2^n x n
Inicializar C[1, 1] a 0  # El coste para el subconjunto {1} y la ciudad 1 es 0

Definir función toIndex(S):
    Inicializar index a 0
    Para cada i en S hacer:
        index <- index + 2^(i - 1)
    Fin Para
    Retornar index
Fin Definir

# Llenar la tabla
Para tamaño desde 2 hasta n hacer:
    subsets <- generar todas las combinaciones
                    de tamaño - 1 elementos de {2, 3, ..., n}
    Para k desde 1 hasta número de columnas de subsets hacer:
        S <- [1] concatenado con subsets[, k]
        subsetIndex <- toIndex(S)
        Para cada j en S[-1] hacer:
            jIndex <- j
            prevSubset <- elementos de S distintos de j
            prevIndex <- toIndex(prevSubset)
            Inicializar min_cost a Infinito
            Inicializar min_city a 0
            Para cada i en prevSubset hacer:
                Si C[prevIndex, i] distinto de Infinito entonces:
                    cost <- C[prevIndex, i] + dist_matrix[i, j]
                    Si cost < min_cost entonces:
                        min_cost <- cost
                        min_city <- i
                    Fin Si
                Fin Si
            Fin Para
            C[subsetIndex, jIndex] <- min_cost
            R[subsetIndex, jIndex] <- min_city
        Fin Para
    Fin Para
Fin Para

fullSubsetIndex <- toIndex(1:n)
Inicializar solution a Infinito
Inicializar min_final_city a 0
Para j desde 2 hasta n hacer:
    cost <- C[fullSubsetIndex, j] + dist_matrix[j, 1]
    Si cost < solution entonces:
        solution <- cost
```

```
            min_final_city <- j
        Fin Si
    Fin Para

    # Reconstruir la ruta óptima
    route <- min_final_city
    subsetIndex <- fullSubsetIndex
    Mientras longitud(route) < n hacer:
        prev_city <- R[subsetIndex, min_final_city]
        route <- prev_city concatenado con route
        subsetIndex <- subsetIndex - 2^(min_final_city - 1)
        min_final_city <- prev_city
    Fin Mientras

    # Añadir la ciudad inicial al final de la ruta
    route <- route concatenado con [1]

    Retornar lista con route = route, distance = solution
Fin Definir

Imprimir "Ruta óptima (Held-Karp):", route
Imprimir "Longitud del recorrido (Held-Karp):", distance
```

Explicación del código: Se inicializa una matriz de distancias y una tabla de costes y rutas. Se llena la tabla generando todas las combinaciones de ciudades y calculando el coste mínimo para cada subconjunto y ciudad final. Para cada subconjunto de ciudades, se evalúa la ruta más corta terminando en una ciudad específica y se actualiza el coste y la ciudad previa en las tablas. Finalmente, se reconstruye la ruta óptima recorriendo las tablas desde la última ciudad hacia atrás, y se retorna la ruta y su coste total.

Los resultados obtenidos son los siguientes:

Ruta óptima (Held-Karp): 1 3 7 4 6 8 5 2 1

Longitud del recorrido (Held-Karp): 945

Tiempo: 1.53 segundos

4.3. Resolución con algoritmos heurísticos

A continuación se muestra la resolución del problema del comerciante de Volvo, esta vez usando algoritmos heurísticos implementados en RStudio. Dentro de las soluciones se calcula la ruta óptima y el coste, que en este caso es la cantidad de kilómetros que debe recorrer el comerciante.

Debido a la complejidad computacional de este problema, especialmente para un gran número de ciudades, se utilizan métodos heurísticos para encontrar soluciones aproximadas en un tiempo razonable.

Esta vez es importante mencionar el uso del paquete "TSP", de RStudio (Hahsler & Hornik, 2023):

Definición 4.2 *TSP: Infraestructura básica y algunos algoritmos para el Problema del Viajante (también conocido como problema del vendedor viajero, TSP, por sus siglas en inglés). El paquete proporciona algunos algoritmos simples y una interfaz para el solucionador de TSP Concorde y*

su implementación de la heurística Chained-Lin-Kernighan. El código de Concorde en sí no está incluido en el paquete y debe obtenerse por separado.

Para abordar el TSP, hemos implementado los siguientes algoritmos heurísticos:

4.3.1. Algoritmo del Vecino más Cercano (KNN)

Este algoritmo selecciona una ciudad inicial y, en cada paso, elige la ciudad no visitada más cercana como el siguiente destino. Este proceso se repite hasta que todas las ciudades han sido visitadas, finalizando el recorrido en la ciudad de inicio.

El pseudocódigo es:

```
Inicializar matriz dist_matrix con los valores dados

Definir función tsp_nearest_neighbor(dist_matrix):
    n <- número de filas de dist_matrix
    Inicializar best_tour a NULL
    Inicializar best_cost a Infinito

    Para start_node desde 1 hasta n hacer:
        Inicializar current_tour a start_node
        Inicializar current_cost a 0
        Inicializar visited a vector de FALSOS de tamaño n
        Inicializar visited[start_node] a VERDADERO
        Inicializar current_node a start_node

        Mientras length(current_tour) < n hacer:
            Inicializar nearest_neighbor a NULL
            Inicializar nearest_distance a Infinito

            Para neighbor desde 1 hasta n hacer:
                Si no visited[neighbor] y
                   dist_matrix[current_node,neighbor] < nearest_distance entonces:
                    nearest_neighbor <- neighbor
                    nearest_distance <- dist_matrix[current_node,neighbor]
                Fin Si
            Fin Para

            current_tour <- current_tour concatenado con nearest_neighbor
            visited[nearest_neighbor] <- VERDADERO
            current_cost <- current_cost + nearest_distance
            current_node <- nearest_neighbor
        Fin Mientras

        # Cerrar el recorrido
        current_cost <- current_cost + dist_matrix[current_node,start_node]
        current_tour <- current_tour concatenado con start_node

        # Reorganizar el recorrido para empezar desde el nodo de inicio (1)
        start_pos <- qué elemento de current_tour es 1
        current_tour <- current_tour[desde start_pos hasta n] concatenado con
```

```
            current_tour[desde 1 hasta start_pos]

        Si current_cost < best_cost entonces:
            best_cost <- current_cost
            best_tour <- current_tour
        Fin Si
    Fin Para

    Retornar lista con best_tour = best_tour, best_cost = best_cost
Fin Definir

Imprimir "Ruta óptima (Vecino más Cercano):", best_tour
Imprimir "Longitud del recorrido (Vecino más Cercano):", best_cost
```

Explicación del código: Se inicializa una matriz de distancias y luego, para cada nodo de inicio posible, se construye una ruta comenzando desde ese nodo y agregando siempre la ciudad no visitada más cercana, hasta visitar todas las ciudades. Al final de cada ruta, se cierra el recorrido regresando al nodo inicial. Se compara el coste de cada ruta con el mejor coste encontrado hasta el momento, actualizando la mejor ruta y su coste si se encuentra una ruta más corta. Finalmente, se retorna la mejor ruta y su coste total.

Los resultados obtenidos son:

Ruta óptima (Vecino más Cercano): 1 2 3 7 5 8 6 4 1

Longitud de recorrido (Vecino más Cercano): 1036

Tiempo: 0.17 segundos

4.3.2. Algoritmo de Colonia de Hormigas (ACO)

Este algoritmo utiliza una población de agentes (hormigas) que construyen soluciones al problema depositando feromonas en los caminos, lo que influye en las decisiones de las hormigas futuras para encontrar rutas cortas de manera colaborativa.

Para este algoritmo es necesario definir previamente algún parámetro. En el caso del ejemplo de los concesionarios Volvo, los parámetros por defecto son los siguientes:

- Número de hormigas = 10
- Número de iteraciones = 100
- Importancia relativa de la feromona = 1
- Importancia relativa de la heurística (visibilidad) = 5
- Tasa de evaporación de la feromona = 0.5
- Constante utilizada en la actualización de la feromona = 100

El pseudocódigo que resuelve el problema mediante el algoritmo de hormigas es el siguiente:

```
Inicializar matriz dist_matrix con los valores dados

Definir función ACO(matriz_distancias,
                    num_hormigas,
                    num_iteraciones,
                    alfa,
                    beta,
```

```
                rho,
                Q)
num_ciudades <- número de filas de matriz_distancias

# Inicializar feromonas
Inicializar feromonas a matriz de unos de tamaño num_ciudades x num_ciudades

# Inicializar visibilidad (inversa de la distancia)
visibilidad = 1 / matriz_distancias

Inicializar mejor_tour a NULL
Inicializar mejor_longitud_tour a Infinito

Para iteración desde 1 hasta num_iteraciones hacer:
    Inicializar todos_tours a lista vacía
    Inicializar todas_longitudes a vector de ceros de tamaño num_hormigas

    Para hormiga desde 1 hasta num_hormigas hacer:
        Inicializar tour a vector de ceros de tamaño num_ciudades + 1
        Inicializar tour[1] a 1  # Empezar en la ciudad 1
        Inicializar visitadas a vector de FALSOS de tamaño num_ciudades
        Inicializar visitadas[1] a VERDADERO

        Para i desde 2 hasta num_ciudades hacer:
            ciudad_actual <- tour[i - 1]
            probabilidades <- (feromonas[ciudad_actual, ]^alfa) *
              (visibilidad[ciudad_actual, ]^beta)
            probabilidades[visitadas] <- 0
            probabilidades <- probabilidades / suma(probabilidades)
            Muestrear una ciudad basada en probabilidades
                y asignarla a próxima_ciudad
            tour[i] <- próxima_ciudad
            visitadas[próxima_ciudad] <- VERDADERO
        Fin Para

        tour[num_ciudades + 1] <- 1  # Terminar en la ciudad 1
        todos_tours[hormiga] <- tour
        longitud_tour <- suma de dist_matrix para los tramos del tour
        todas_longitudes[hormiga] <- longitud_tour

        Si longitud_tour < mejor_longitud_tour entonces:
            mejor_longitud_tour <- longitud_tour
            mejor_tour <- tour
        Fin Si
    Fin Para

    # Actualización de feromonas
    feromonas <- (1 - rho) * feromonas
    Para hormiga desde 1 hasta num_hormigas hacer:
```

```
            tour <- todos_tours[hormiga]
            longitud_tour = todas_longitudes[hormiga]
            Para i desde 1 hasta num_ciudades hacer:
                feromonas[tour[i], tour[i + 1]] <-
                  feromonas[tour[i], tour[i + 1]] + Q / longitud_tour
                feromonas[tour[i + 1], tour[i]] <-
                  feromonas[tour[i + 1], tour[i]] + Q / longitud_tour
            Fin Para
        Fin Para
    Fin Para

    Retornar lista con mejor_tour = mejor_tour,
                        mejor_longitud_tour = mejor_longitud_tour
Fin Definir

Imprimir "Ruta óptima (Colonia de Hormigas):", mejor_tour
Imprimir "Longitud del recorrido (Colonia de Hormigas):", mejor_longitud_tour
```

Explicación del código: Se inicializa una matriz de distancias y feromonas, y se calcula la visibilidad (inversa de las distancias). Durante varias iteraciones, cada hormiga construye un tour comenzando desde la ciudad inicial, seleccionando la siguiente ciudad basada en probabilidades que dependen de las feromonas y la visibilidad. Al final de cada iteración, se actualizan las feromonas en función de la calidad de los tours encontrados. Se compara la longitud de cada tour con la mejor longitud encontrada hasta el momento, actualizando el mejor tour y su longitud si se encuentra un tour más corto. Finalmente, se retorna el mejor tour y su longitud total.

Los resultados obtenidos son:

Ruta óptima (Colonia de Hormigas): 1 2 3 7 4 6 8 5 1

Longitud de recorrido (Colonia de Hormigas): 947

Tiempo: 35.63 segundos

4.3.3. Algoritmos de Inserción

4.3.3.1. Nearest Insertion

Este algoritmo empieza con un subcircuito pequeño y agrega iterativamente la ciudad más cercana no incluida en el subcircuito en la posición que minimice la distancia total.

Para resolver este algoritmo en R, es necesario instalar el paquete TSP.

El pesudocódigo es el siguiente:

```
Inicializar matriz dist_matrix con los valores dados

# Convertir la matriz de distancias en un objeto TSP
tsp_instance <- TSP(dist_matrix)

# Resolver usando Nearest Insertion
nearest_insertion_solution <-
  solve_TSP(tsp_instance, method = "nearest_insertion")

# Obtener la ruta y agregar la ciudad inicial (1) al final
ruta_nearest_insertion <- as.integer(nearest_insertion_solution)
```

```
ruta_nearest_insertion <- [1] concatenado con
    ruta_nearest_insertion[ruta_nearest_insertion distinto de 1]
    concatenado con [1]

Inicializar longitud_recorrido_nearest_insertion a 0
Para i desde 1 hasta longitud(ruta_nearest_insertion) - 1 hacer:
    longitud_recorrido_nearest_insertion <-
      longitud_recorrido_nearest_insertion +
      dist_matrix[ruta_nearest_insertion[i], ruta_nearest_insertion[i + 1]]
Fin Para

Imprimir "Ruta óptima (Nearest Insertion):", ruta_nearest_insertion
Imprimir "Longitud del recorrido (Nearest Insertion):",
    longitud_recorrido_nearest_insertion
```

Explicación del código: Se convierte la matriz de distancias en una instancia del Problema del Viajante (TSP) y se resuelve usando el método de Inserción Más Cercana. Después de obtenerse la solución, se ajusta la ruta para que empiece y termine en la ciudad inicial, se calcula la longitud total del recorrido sumando las distancias entre ciudades consecutivas en la ruta ajustada y finalmente se imprime la ruta óptima y su longitud total.

Los resultados obtenidos son:

Ruta óptima (Nearest Insertion): 1 4 6 8 5 2 3 7 1

Longitud del recorrido (Nearest Insertion): 999

Tiempo: 0 segundos

4.3.3.2. Farthest Insertion

Similar al anterior, pero inserta la ciudad más lejana no incluida en el subcircuito en cada iteración, en la posición que minimice la distancia total.

El pesudocódigo usado en R es el siguiente:

```
Inicializar matriz dist_matrix con los valores dados

# Convertir la matriz de distancias en un objeto TSP
tsp_instance <- TSP(dist_matrix)

# Resolver usando Farthest Insertion
farthest_insertion_solution <-
  solve_TSP(tsp_instance, method = "farthest_insertion")

# Obtener la ruta y agregar la ciudad inicial (1) al final
ruta_farthest_insertion <- as.integer(farthest_insertion_solution)
ruta_farthest_insertion <- [1] concatenado con
    ruta_farthest_insertion[ruta_farthest_insertion distinto de 1]
    concatenado con [1]

Inicializar longitud_recorrido_farthest_insertion a 0
Para i desde 1 hasta longitud(ruta_farthest_insertion) - 1 hacer:
    longitud_recorrido_farthest_insertion <-
```

```
      longitud_recorrido_farthest_insertion +
      dist_matrix[ruta_farthest_insertion[i], ruta_farthest_insertion[i + 1]]
Fin Para

Imprimir "Ruta óptima (Farthest Insertion):", ruta_farthest_insertion
Imprimir "Longitud del recorrido (Farthest Insertion):",
    longitud_recorrido_farthest_insertion
```

Explicación del código: Se convierte la matriz de distancias en una instancia del Problema del Viajante (TSP) y se resuelve usando el método de Inserción Más Lejana. Luego, se ajusta la ruta para que comience y termine en la ciudad inicial, se calcula la longitud total del recorrido sumando las distancias entre las ciudades consecutivas en la ruta ajustada y finalmente se imprime la ruta óptima y su longitud total.

Los resultados obtenidos son:

Ruta óptima (Farthest Insertion): 1 4 2 5 8 6 3 7 1

Longitud del recorrido (Farthest Insertion): 1100

Tiempo: 0 segundos

Capítulo 5

Aplicación Web

En esta sección se describe cómo se puede hacer una aplicación web desarrollada en R utilizando el paquete Shiny para resolver el Problema del Viajante mediante los diferentes algoritmos utilizados. La aplicación permite cargar una matriz de distancias desde un archivo de Excel y seleccionar el algoritmo deseado para resolver el problema.

La aplicación devuelve el recorrido más óptimo, su longitud y el tiempo de ejecución del algortimo elegido.

A continuación, se explica paso a paso el proceso de desarrollo de esta aplicación.

5.1. Paquetes instalados

Para comenzar, se instalan y cargan los paquetes necesarios:

- "shiny": para crear aplicaciones web interactivas.
- "gtools": proporciona funciones útiles para el manejo de datos y operaciones matemáticas.
- "TSP": contiene herramientas específicas para resolver el Problema del Viajante.
- "readxl": permite leer archivos Excel.

5.2. Interfaz de Usuario

Definición 5.1 *La interfaz de usuario, conocida también como IU, es el medio visual que permite la comunicación entre un usuario y una máquina y comprende todos los puntos de contacto entre el usuario y el equipo. En otras palabras, es el puente que conecta a los usuarios con la tecnología.*

En la interfaz aparece el título de la aplicación, una descripción de la aplicación, las opciones de selección de algoritmos y un botón para cargar archivos. Además, una vez se hayan cargado los datos necesarios, aparece un área donde se mostrarán los resultados.

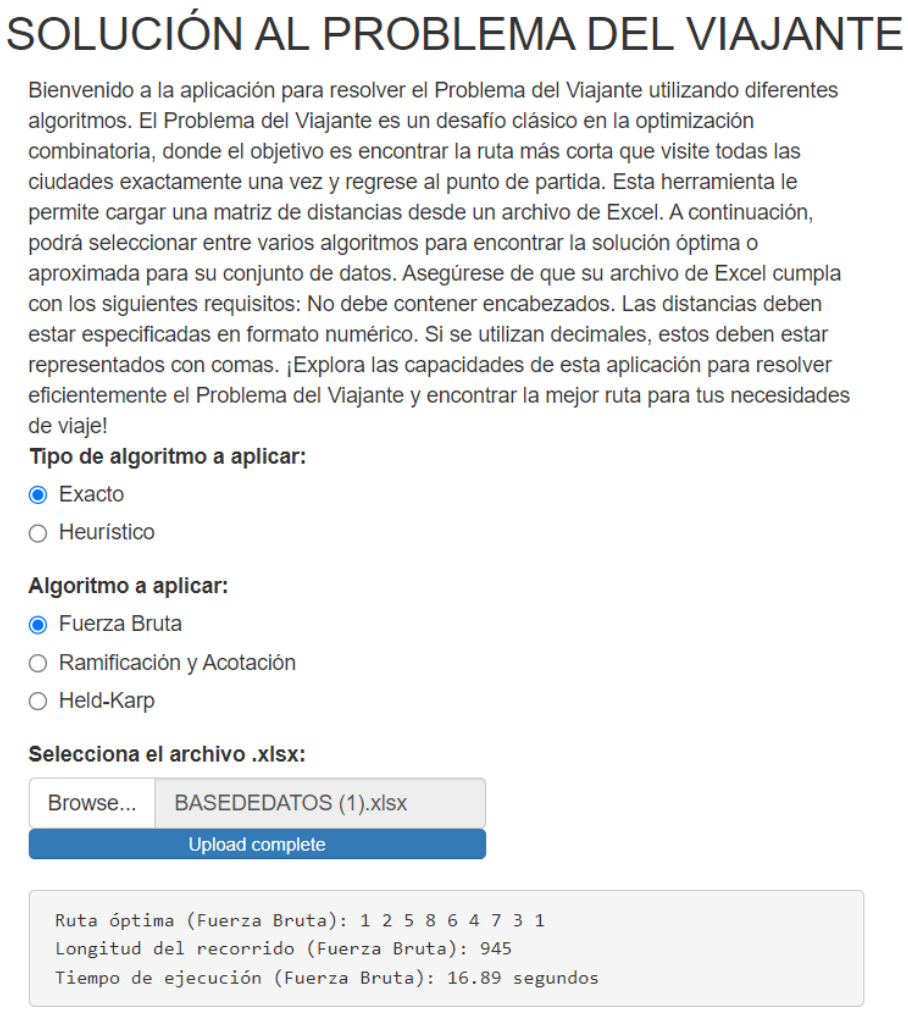

Figura 5.1: Interfaz de la aplicación web creada

5.3. Algoritmos de Solución

La aplicación ofrece varios algoritmos para resolver el TSP:

1. Exactos:
   - Fuerza Bruta: Evalúa todas las permutaciones posibles de las ciudades.
   - Ramificación y Acotación: Utiliza un enfoque de búsqueda con poda para reducir el espacio de búsqueda.
   - Held-Karp: Implementa un algoritmo de programación dinámica para resolver el TSP.
2. Heurísticos:
   - Vecino más Cercano: Selecciona la ciudad más cercana no visitada en cada paso.
   - Colonia de Hormigas: Simula el comportamiento de las hormigas para encontrar rutas óptimas.
   - Inserción (Nearest y Farthest): Inserta ciudades en la ruta basándose en criterios de proximidad.

Limitaciones

A pesar de ser una aplicación que funciona para resolver problemas de tipo TSP, sólo esta probado para tamaños de datos de pocas instancias, por lo que en una matriz de distancia tendríamos problemas en cuanto a la complejidad computacional.

Otra de las limitaciones que presenta esta aplicación es la forma de meter los datos, ya que solo lee archivos en formato Excel que cumplan con los siguientes requisitos:

1. No debe contener encabezados.
2. Las distancias deben estar especificadas en formato numérico.
3. Si se utilizan decimales, estos deben estar representados con comas.

A pesar de las limitaciones, la aplicación desarrollada proporciona una herramienta interactiva y fácil de usar para resolver el Problema del Viajante utilizando distintos algoritmos exactos y heurísticos.

Capítulo 6

Conclusión

RESULTADOS

Tabla 6.1: Resultados de los algoritmos exactos para el Problema del Viajante

Algoritmo	**Mejor Ruta**	**Coste**	**Tiempo**
Fuerza Bruta	1 2 5 8 6 4 7 3 1	945	30.56 segundos
Ramificación y Acotación	1 2 5 8 6 4 7 3 1	945	140.33 segundos
Held-Karp	1 3 7 4 6 8 5 2 1	945	1.53 segundos

Tabla 6.2: Resultados de los algoritmos heurísticos para el Problema del Viajante

Algoritmo	**Mejor Ruta**	**Coste**	**Tiempo**
KNN	1 2 3 7 5 8 6 4 1	1036	0.17 segundos
Colonia de hormigas	1 2 3 7 4 6 8 5 1	947	35.63 segundos
Nearest Insertion	1 4 6 8 5 2 3 7 1	999	0 segundos
Farthest Insertion	1 4 2 5 8 6 3 7 1	1100	0 segundos

Evaluación de Algoritmos Exactos

Los algoritmos exactos (Fuerza Bruta, Ramificación y Acotación, y Held-Karp) lograron encontrar la ruta óptima con un coste de 945.

El algoritmo de Fuerza Bruta, aunque es el más simple conceptualmente, fue el más rápido en este caso específico (30.56 segundos) debido al pequeño tamaño del problema (8 ciudades). Sin embargo, su complejidad exponencial lo hace impracticable para instancias mayores.

La Ramificación y Acotación, aunque también exacta, requirió un tiempo significativamente mayor (140.33 segundos), mostrando cómo el aumento en la eficiencia del algoritmo puede venir a costa de tiempos de cálculo mayores en algunos casos específicos.

El algoritmo de Held-Karp, basado en programación dinámica, se mostró eficiente con un tiempo de ejecución de 1.53 segundos, siendo una opción viable para tamaños moderados del problema.

Evaluación de Algoritmos Heurísticos

Los algoritmos heurísticos (KNN, Colonia de Hormigas, Nearest Insertion y Farthest Insertion) proporcionaron soluciones aproximadas de manera eficiente, aunque con ligeras desviaciones del coste óptimo.

El algoritmo KNN es rápido y sencillo de implementar, a pesar de ello no garantiza una solución óptima debido al alto coste, 1036, en comparación con el resto de algoritmos.

La Colonia de Hormigas produjo una solución casi óptima con un coste de 947 en 35.63 segundos, demostrando ser un algoritmo robusto y eficiente para obtener soluciones de alta calidad en un tiempo razonable.

Los métodos Nearest Insertion y Farthest Insertion ofrecieron soluciones con un coste de 999 y 1100 respectivamente, siendo más rápidos en términos de tiempo de ejecución (0 segundos), pero menos precisos en comparación con la Colonia de Hormigas.

Rendimiento y Complejidad Computacional

La investigación y análisis realizados en este trabajo han demostrado que los algoritmos exactos proporcionan la solución óptima para el TSP, aunque pueden tener tiempos de ejecución elevados para problemas grandes. El algoritmo Held-Karp destacó por su eficiencia en términos de tiempo en comparación con otros métodos exactos. Por otro lado, los algoritmos heurísticos son significativamente más rápidos y pueden ofrecer soluciones cercanas a la óptimo. La Colonia de Hormigas mostró ser el mejor algoritmo de entre los heurísticos en términos de calidad de la solución, aunque con un tiempo de ejecución mayor que el resto de ellos.

La elección del algoritmo depende del tamaño del problema y de la necesidad de precisión frente a la velocidad. Para problemas de pequeños a medianos donde se requiere precisión, Held-Karp es una excelente opción. Para problemas grandes donde se necesita una solución rápida, los heurísticos como KNN o Colonia de Hormigas son más apropiados.

El programa desarrollado en R ha permitido la implementación y ejecución eficiente de estos algoritmos, facilitando un análisis exhaustivo de su rendimiento y complejidad computacional.

Bibliografía

Alfaro, L. (2020). *Ciclo hamiltoniano óptimo en un grafo (problema del viajante)*. Universidad de Zaragoza. Facultad de Ciencias.

Alvarez-Nuñez, M. (2013). *Teoría de grafos*. Universidad del Bío-Bío. Escuela de Pedagogía en Educación Matemática (Chile).

Applegate, D., Bixby, R., Chvatal, V., & Cook, W. (2003). Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Mathematical programming*, *97*, 91-153.

Asociación Nacional de Estudiantes de Matemáticas. (2018). *El problema del viajante* [Accessed: 01-07-2024]. https://www.anem.es/web/2018/09/el-problema-del-viajante/#:~:text=Hoy%20en%20d%C3%ADa%2C%20el%20TSP,exacta%20en%20un%20tiempo%20razonable.

Ball, W. W. R., & Coxeter, H. S. M. (1987). Mathematical recreations and essays (13th edition). *The Mathematical Gazette*, *72*(460), 428. https://doi.org/10.1017/S0025557200142585

Cabezas, X. (2014). *Problemas del milenio: P vs np*. Amarun.

Chaves, C., & Miloca, S. A. (2018). Algorítmo Branch-and-Bound e aplicação ao problema do caixeiro viajante. *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*, *6*(1).

Cook, W., Cunningham, W., Pulleyblank, W., & Schrijver, A. (1994). Combinatorial optimization. *10*, 75-93.

Crehuet-Lucas, I. (2022). *El problema del viajante con grafos*. Trabajo Fin de Grado. Grado en Matemáticas. Universidad de Valladolid.

Crilly, T. (2005). Landmark Writings in Western Mathematics 1640-1940. *Papers on dimension theory (1923–1926)*, 844-855.

Dantzig, G., Fulkerson, R., & Johnson, S. (2003). Solution of a large scale traveling salesman problem. *50 Years of Integer Programming 1958–2008*, 7.

Dantzig, G., Orden, A., & Wolfe, P. (1955). The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, *5*(2), 183-195.

Dirac, G. A. (1952). Some theorems on abstract graphs. *Proceedings of the London Mathematical Society*, *3*(1), 69-81.

Domínguez-Pérez, J. (2020). *Apuntes para un curso de Matemática Discreta y Optimización*. Departamento de Matemáticas, Universidad de Salamanca.

Dorigo, M., & Gambardella, L. (1997). Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, *1*(1), 53-66.

Euler, L. (1741). Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 128-140.

Flood, M., & Savage, L. (1948). A Game theoretic study of the tactics of area defense. *RAND Research Memorandum*, *51*.

Garey, D. (1979). *Johnson, Computers and Intractability. A Guide to the Theory of NP-Completeness*. WH Freeman; Company, New York.

Godsil, C., & Royle, G. F. (2001). *Algebraic graph theory* (Vol. 207). Springer Science & Business Media.

Golden, B., Bodin, L., Doyle, T., & Stewart Jr, W. (1980). Approximate traveling salesman algorithms. *Operations research*, *28*(3-part-ii), 694-711.

González-Santander, G. (2020). *Tres métodos diferentes para resolver el problema del viajante* [Accessed: 01-07-2024]. https://baobabsoluciones.es/blog/2020/10/01/problema-del-viajante/

Hahsler, M., & Hornik, K. (2007). TSP-Infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, *23*(2), 1-21.

Hahsler, M., & Hornik, K. (2023). *TSP: Traveling Salesperson Problem (TSP)* [R package version 1.2-4]. https://CRAN.R-project.org/package=TSP

Hamilton, W. (1858). Account of the icosian calculus. *Proceedings of the Royal Irish Academy*, *6*(1858), 415-416.

Núñez, V., Perez, J., Perez, A., et al. (2004). Siete puentes, un camino: Königsberg. *Suma: revista sobre la enseñanza y aprendizaje de las matemáticas, 45, 69-78.*

Ore, O. (1960). A note on hamiltonian circuits. *American Mathematical Monthly*, *67*, 55.

Pérez-de Vargas-Moreno, B. (2015). *Resolución del Problema del Viajante de Comercio (TSP) y su variante con Ventanas de Tiempo (TSPTW) usando métodos heurísticos de búsqueda local*. Escuela de Ingenierías Industriales. Universidad de Valladolid.

Pérez-Jiménez, M. (2019). *Apuntes para un curso de Teoría de la Complejidad Computacional*. Grupo de Investigación en Computación Natural. Dpto. Ciencias de la Computación e Inteligencia Artificial. Universidad de Sevilla.

Reinelt, G. (1992). Fast heuristics for large geometric traveling salesman problems. *ORSA Journal on computing*, *4*(2), 206-217.

Rosen, K. (2012). *Discrete Mathematics and Its Applications. Seventh Edition*. McGraw-Hill.

Rosenkrantz, D., Stearns, R., & Lewis II, P. (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, *6*(3), 563-581.

Serna, S. F., Elena et al. (2021). Relajación y heurística Lagrangiana. Aplicación a un problema de optimización combinatoria.

Tello, E. (2018). Algoritmos de búsqueda exhaustiva. *CINVESSTAV-Tamaulipas*.

Turing, A. (2004). *The essential turing*. Oxford University Press.

Universidad de Cadiz. (2021). *Problema del Viajante de Comercio* [Accessed: 01-07-2024]. https://knuth.uca.es/moodle/mod/resource/view.php?id=4130

Voigt, B. (1831). Der Handlungsreisende, wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein. *Commis-Voageur, Ilmenau*, 69-72.