Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

EL PROBLEMA DEL VIAJANTE DE COMERCIO

Grado en Ingeniería Mecánica

ALUMNO: Alejandro García Sanz

DIRECTOR: Rodrigo Mariño Villar

CURSO ACADÉMICO: 2024-2025

Universidade Vigo

Centro Universitario de la Defensa
en la Escuela Naval Militar

TRABAJO FIN DE GRADO

EL PROBLEMA DEL VIAJANTE DE COMERCIO

Grado en Ingeniería Mecánica

Intensificación en Tecnología Naval

Cuerpo General

Universidade Vigo

RESUMEN

El presente Trabajo Fin de Grado aborda el Problema del Viajante de Comercio (TSP), un desafío clásico de optimización combinatoria con amplias aplicaciones en logística, industria, inteligencia artificial o incluso el ámbito militar. Se formula el problema desde una perspectiva matemática y computacional, destacando su clasificación como problema NP-hard, lo que implica que su resolución exacta se vuelve inabordable conforme crece el número de nodos. El trabajo analiza en profundidad métodos exactos como la fuerza bruta y la programación dinámica, aproximaciones como el algoritmo de Christófides y métodos heurísticos cómo el método de inserción más cercana. También se mencionan métodos metaheurísticos, como el recocido simulado o la colonia de hormigas. Se presta especial atención a la aplicabilidad práctica de algunos de estos algoritmos, evaluando su rendimiento en escenarios reales con apoyo de la herramienta SageMath. Como caso práctico, se modela una misión de patrulla aérea en la zona del golfo de Cádiz y mar de Alborán, aplicando diferentes técnicas para minimizar la distancia recorrida. Este estudio busca proporcionar una visión clara sobre la efectividad de cada enfoque, sirviendo como guía para futuras implementaciones del TSP en contextos operativos más complejos, incluyendo el entorno de las Fuerzas Armadas.

PALABRAS CLAVE

Optimización, Grafo, Circuito Hamiltoniano, Complejidad computacional, NP-hard.

AGRADECIMIENTOS

A mi familia, amigos, pareja y a mi querido León.

CONTENIDO

Contenido 1

Índice de Figuras 3

Índice de Tablas 5

1 Introducción y objetivos 6

1.1 Introducción 6

1.2 Motivación 7

1.3 Objetivos 7

2 Estado del arte 9

2.1 Marco histórico 9

2.1.1 Los orígenes en la teoría de grafos. Euler y los puentes de Königsberg (1736) 9

2.1.2 William Rowan Hamilton y los ciclos hamiltonianos (1857-1862). 10

2.1.3 Karl Menger y el Problema del Mensajero (1920s): un antecedente directo del TSP 10

2.1.4 Formalización del TSP (1930-1950). Origen del problema en la planificación de rutas 11

2.1.5 El estudio de Dantzig, Fulkerson y Johnson (1954): el primer enfoque computacional 12

2.1.6 Planos de Corte 14

2.1.7 Método de ramificación y acotación (Branch & Bound) 14

2.1.8 Origen de la clasificación del problema 15

2.1.9 Avances computacionales y algoritmos exactos 16

2.1.10 Métodos heurísticos, aproximados y metaheurísticos 17

2.2 Complejidad del problema. 19

2.2.1 Implicaciones 20

2.2.2 Soluciones según el número de nodos 20

2.2.3 Computación temporal y notación asintótica 21

3 Desarrollo del TFG 23

3.1 Introducción 23

3.1.1 Formulación matemática TSP y su relación con la teoría de grafos. 23

3.2 Métodos exactos 24

3.2.1 Fuerza bruta 24

3.2.2 Algoritmo de Belhman-Held-Karp 26

3.3 Árbol de expansión mínima (MST) 27

3.4 Métodos aproximados 29

3.4.1 Cálculo de una cota inferior 30

3.4.2 Algoritmo de Christófides 31

3.5 Métodos heurísticos 33

3.5.1 Cota superior. Método de inserción más cercana 33
3.5.2 Método del árbol 35
4 Resultados / Validación / Prueba 38
4.1 Resolución por métodos exactos 41
4.2 Resolución por métodos aproximados 42
4.2.1 Cálculo de una cota inferior. Método del $1-árbol$ 42
4.2.2 Algoritmo de Christófides 43
4.3 Resolución por métodos heurísticos 48
4.3.1 Cota superior. Método de inserción más cercana 48
4.3.2 Método del árbol 53
5 Conclusiones y líneas futuras 56
6 Bibliografía 58
Anexo I: Impacto económico y ambiental 61
Anexo II: Herramienta SageMath para apoyar resolución 63

ÍNDICE DE FIGURAS

Figura 1. Abstracción del problema de los Siete Puentes de Königsberg. [25]……9

Figura 2. 30 posibles soluciones para el Juego Icosiano. [29]……10

Figura 3. Simplex m-dimensional. [34]……12

Figura 4. Matriz del problema de 49 ciudades. [35]……..13

Figura 5. Ruta óptima hallada para matriz de la figura 4. [35]…..……13

Figura 6. Mapa seccionado con un método precario de Branch & Bound para su resolución. Mismo problema que las figuras 3 y 4. [35]……..15

Figura 7. TSP resuelto mediante métodos exactos a través de los años. [1]……..…16

Figura 8. Gráfica exponencial de los avances en resolución de métodos exactos. [1]…….…...17

Figura 9. Ejemplo de 400 ciudades con 4 “temperaturas” distintas en el método del recocido simulado para resolución del TSP. [44]……...18

Figura 10. El comportamiento de la colonia termina por obtener el camino más corto entre dos puntos. [46]……..19

Figura 11. Ejemplo del MST hallado mediante el algoritmo de Prim. [2]……..…….27

Figura 12. Ejemplo sencillo del método del 1-árbol. [9]……31

Figura 13. Elaboración propia curso 2025. Ciclo inicial C_2……...…34

Figura 14. Elaboración propia curso 2025, herramienta OpenStreetMap. Grafo superpuesto al mapa de la zona……...……..38

Figura 15. Elaboración propia curso 2025 con la herramienta SageMath. Grafo que expone la situación de los nodos y la posible conectividad de cada uno de ellos con el resto de nodos…….40

Figura 16. Elaboración propia curso 2025. Generado con SageMath. Este grafo muestra el circuito hamiltoniano correspondiente a la solución exacta de nuestro TSP……..……..41

Figura 17. Elaboración propia curso 2025 a través de la herramienta SageMath. Árbol de mínima expansión utilizado para obtener una cota inferior con el método del 1-árbol……42

Figura 18. Elaboración propia curso 2025 a través de la herramienta SageMath. Ilustración del resultado del método del 1-árbol……..……43

Figura 19. Elaboración propia curso 2025 con el programa SageMath. Árbol de mínima expansión del ejercicio propuesto……44

Figura 20. Elaboración propia curso 2025 con el programa SageMath. Emparejamiento perfecto mínimo en nodos impares……45

Figura 21. Elaboración propia curso 2025 con programa SageMath. Multigrafo: MST + Emparejamiento Perfecto Mínimo……...46

Figura 22. Elaboración propia curso 2025 con programa SageMath. Circuito euleriano……...47

Figura 23. Elaboración propia curso 2025 con programa SageMath. Circuito hamiltoniano solución subóptima mediante método de Christófides……...48

Figura 24. Elaboración propia curso 2025 a través de SageMath. Ciclo de partida para el método de inserción más cercana……..49

Figura 25. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Ceuta…49

Figura 26. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Málaga.50

Figura 27. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Peñón de Vélez……50

Figura 28. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Melilla..51

Figura 29. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Chafarinas……51

Figura 30. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Isla de Alborán……52

Figura 31. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Almería……52

Figura 32. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Cartagena…….53

Figura 33. Elaboración propia curso 2025 a través de SageMath con ayuda de IA generativa. La IA se ha empleado para generar la línea roja curva que simula la duplicidad de la arista del MST para favorecer la visualización. Ciclo euleriano con aristas duplicadas…….……..54

Figura 34. Elaboración propia curso 2025 con programa SageMath. Ciclo hamiltoniano obtenido con el método del árbol……55

ÍNDICE DE TABLAS

Tabla 1. Elaboración propia curso 2025. Relación entre el número de nodos y el número de circuitos hamiltonianos solución de ese problema.....21

Tabla 2. Elaboración propia curso 2025. Comparación de métodos con sus respectivas complejidades temporales……...22

Tabla 3. Elaboración propia curso 2025. Resumen comparativo de los métodos para generar MST. [2]……29

Tabla 4. Matriz de distancias ejemplo 3.3.3.6……33

Tabla 5. Elaboración propia curso 2025 con datos sacados de la herramienta Open CPN. Los valores se muestran en grados siendo positivos los valores de latitud norte y los valores de longitud este……40

Tabla 6. Elaboración propia curso 2025. Las distancias se expresan en millas náuticas. Los datos se han tomado mediante SageMath a través de las coordenadas……...61

1 INTRODUCCIÓN Y OBJETIVOS

"Empty your mind, be formless, shapeless, like water. Now you put water into a cup, it becomes the cup, you put it into a bottle, it becomes the bottle, you put it in a teapot, it becomes the teapot. Now water can flow, or it can crash. Be water my friend."

Bruce Lee.

1.1 Introducción

El presente trabajo ha sido enfocado en el desarrollo y estudio del Problema del viajante de comercio conocido internacionalmente por sus siglas en inglés como TSP (Travelling Salesman Problem), se nombra como TSP en numerosas ocasiones a lo largo del trabajo. Este problema es uno de los más importantes y ampliamente estudiados dentro del ámbito de la optimización, destacando por su relevancia tanto teórica como práctica. Además, el TSP tiene una presencia significativa en el campo de la informática, dónde se utiliza como caso de estudio para analizar la complejidad computacional y el diseño de algoritmos eficientes.

La formulación del problema, es en sí bastante simple, y no resulta complejo de comprender para un público general. Imaginaremos un comerciante que pretende recorrer una serie de ciudades para llevar su mercancía a todas ellas y de esta forma cumplir con su demanda de pedidos en los puntos requeridos. La premisa principal que debe tener en cuenta este comerciante para el diseño de su ruta es que debe recorrer todas las ciudades una única vez y retornar a la de origen sin haber repetido estancia en ninguna de éstas. El objetivo del comerciante debe ser el de realizar un itinerario consumiendo el menor número de recursos posible; como por ejemplo kilómetros o euros de combustible gastados. La aparente sencillez del problema se encuentra alejada de la realidad, pues su resolución óptima resulta ser de una enorme complejidad. Esta aparente paradoja entre la simplicidad conceptual y la dificultad práctica se debe a la naturaleza combinatoria del problema, que implica evaluar un número factorial de posibles rutas a medida que aumenta el número de ciudades. Esta característica, hace que el TSP adquiera la denominación de problema NP-hard. Esto significa que, si tomásemos como referencia el peor de los casos posibles a la hora de seleccionar el itinerario, el tiempo requerido para encontrar la solución óptima crecería exponencialmente conforme el tamaño del grafo fuese aumentando también, haciéndolo computacionalmente intratable para problemas grandes mediante métodos exactos. En otras palabras; esto quiere decir, que conforme el número de ciudades crece, el tiempo que conlleva la prueba de todas las rutas posibles para así determinar la más corta, se vuelve inasumible e inabarcable en un tiempo que se pudiese llegar a tomar cómo razonable. Acorde a esto, entra también a jugar como factor importante la cantidad de tiempo que realmente merece la pena invertir para hallar la verdadera ruta más corta, y si este tiempo es menor que el que se emplearía para conseguir una ruta que no garantice ser la más óptima pero que se halle de una forma más rápida, usando diversos métodos de aproximación que posteriormente se han nombrado y tratado en este trabajo.

El TSP no solo representa un desafío teórico en el estudio de la complejidad algorítmica, sino que también sirve como un banco de pruebas para el desarrollo y la aplicación de técnicas avanzadas de optimización, tanto exactas como heurísticas. Su estudio ha impulsado avances significativos en áreas como la programación lineal, la inteligencia artificial y las metaheurísticas, consolidándose como un pilar importante en la investigación operativa y la ciencia de la computación.

1.2 Motivación

El TSP, en absoluto es un problema únicamente relevante en el campo de las matemáticas y la computación. Su resolución de una forma eficiente tiene un impacto de gran importancia en múltiples sectores. Multitud de empresas especializadas en logística y distribución buscan incesantemente un método para minimizar costos de transporte haciendo que sus rutas resulten paulatinamente más optimizadas. Al mismo tiempo, las industrias implicadas en procesos de manufactura necesitan una mejora considerable en sus procesos de ensamblaje y en sus circuitos electrónicos al mismo tiempo que realizan una exhaustiva planificación en las tareas de diseño de líneas de producción eficientes.

En cuanto al apartado tecnológico, cabe destacar fundamentalmente que el estudio de algoritmos para facilitar la aproximación y comprensión del TSP ha impulsado de forma activa el desarrollo de técnicas para la modernización de sistemas de inteligencia artificial y aprendizaje automático. Este factor es muy importante para los avances conseguidos en campos tales como la conducción autónoma y la navegación por medio de robots. El TSP, además, es uno de los mayores nichos de estudio para la solución de problemas más complejos, cómo por ejemplo el Vehicle Routing Problem (VRP) o el Pick-Up and Delivery Problem (PDP) los cuales radican de éste, con la diferencia de que se les añaden un número más elevado de restricciones.

En cuanto al ámbito de las Fuerzas Armadas se refiere, más concretamente en la Armada, el estudio de este problema resulta de gran interés. La Armada se enfrenta a innumerables retos logísticos a diario por su más que reconocible carácter errante. La optimización de las vías de aprovisionamiento tanto en puerto base como en territorios de ultramar es vital para el próspero devenir de las unidades. Además, una selección acertada en el itinerario en tránsitos con distintas escalas programadas puede repercutir positivamente en factores como el tiempo, los costes o el almacenaje de víveres. Esto es por supuesto extrapolable a aplicaciones en los otros ejércitos, cómo el Ejército del Aire a la hora de programar patrullas aéreas o vuelos de reconocimiento con el mismo fin de optimizar costes, tiempos y consumo de recursos. Cabe resaltar también los posibles beneficios que imprime a los sistemas de comunicaciones y sistemas de armas y combate debido a la importante repercusión que el TSP entraña en sistemas de computación, ayudando con la implementación y optimización de enlaces.

Dada la dificultad computacional del TSP y la necesidad de obtener soluciones óptimas o cercanas a este resultado en tiempos razonables, este trabajo pretende analizar y comparar distintos enfoques para lograr hallar una aproximación aceptable. Con esto, se busca aportar una visión clara sobre las ventajas y limitaciones de cada método, proporcionando una referencia útil para futuras aplicaciones del TSP en entornos reales y a ser posible, a ámbitos cercanos a la profesión militar.

1.3 Objetivos

Para alcanzar el propósito de este trabajo, se ha buscado en primer lugar describir y formalizar matemáticamente el TSP, estableciendo su formulación en términos de teoría de grafos y optimización matemática, así como analizar su complejidad computacional. A partir de esta base teórica, se ha procedido a analizar diferentes métodos de resolución:

En primer lugar, se han tomado en cuenta los algoritmos exactos; que pese a su baja eficiencia sobre todo cuando se trata de problemas de gran tamaño, tratan de encontrar la solución óptima salvando el problema de su alto y exponencialmente creciente coste computacional.

También se han estudiado métodos aproximados. Los cuáles a cambio de ofrecer un coste computacional más asequible, arrojan un resultado con un error determinado ε.

Se han probado también algunos de los métodos heurísticos más significativos. Este tipo de métodos, buscan soluciones rápidas, aunque no necesariamente óptimas. No garantizan una tasa de error determinada.

También, se han mencionado métodos metaheurísticos, los cuales combinan estrategias avanzadas para mejorar la calidad de la solución sin incurrir en los elevados tiempos de cálculo de los métodos exactos.

Tras la exposición de los diferentes métodos de resolución a emplear, se ha llevado a cabo la implementación y comparación de distintos algoritmos, evaluando su eficiencia en términos de calidad de la solución y tiempo de computación con el fin de determinar cuál de ellos es más adecuado para diferentes tamaños y configuraciones del problema. Esta prueba se ha llevado a cabo con un escenario aplicado al ámbito de las fuerzas armadas, colocando el foco del trabajo sobre una patrulla aérea que busca realizar un control de las zonas de interés costero en las inmediaciones del golfo de Cádiz y mar de Alborán. Dicha patrulla busca recorrer la mínima distancia posible y por tanto disminuir el consumo de combustible.

2 ESTADO DEL ARTE

2.1 Marco histórico

El problema del viajante de comercio tiene raíces en conceptos matemáticos que datan de los siglos XVIII y XIX. Aunque su formulación moderna surgió en el siglo XX, sus principios fundamentales pueden rastrearse hasta problemas clásicos de teoría de grafos y combinatoria.

2.1.1 Los orígenes en la teoría de grafos. Euler y los puentes de Königsberg (1736)

El primer avance significativo que sentó las bases para el estudio de rutas óptimas proviene del matemático Leonhard Euler (1707-1783) con su trabajo sobre los Siete Puentes de Königsberg en 1736. Euler demostró que no era posible recorrer los puentes de la ciudad de Königsberg (actualmente Kaliningrado, Rusia) cruzando cada uno exactamente una vez y volviendo al punto de inicio.[24]

Para demostrarlo, Euler representó el problema mediante un grafo, donde cada región terrestre se correspondía con un nodo y cada puente con una arista. Con ello, introdujo los conceptos de caminos y ciclos en grafos, que hoy en día son fundamentales para la teoría de rutas. Este trabajo marcó el nacimiento de la teoría de grafos y estableció las condiciones para la existencia de un ciclo Euleriano.

Aunque el problema del viajante de comercio no es un problema euleriano, ya que el TSP busca optimizar la visita a nodos en lugar de aristas, los métodos desarrollados por Euler proporcionaron las primeras herramientas para el análisis de recorridos en grafos. Esto resultó un acontecimiento precursor para futuros estudios en optimización de rutas.

Figura 1. Abstracción del problema de los Siete Puentes de Königsberg. [25]

2.1.2 William Rowan Hamilton y los ciclos hamiltonianos (1857-1862).

El siguiente gran avance en la formalización matemática del TSP ocurrió en el siglo XIX con el trabajo de William Rowan Hamilton (1805-1865) y Thomas Penyngton Kirkman (1806-1895).

Hamilton fue un matemático irlandés que, en la década de 1850, desarrolló el concepto de caminos y ciclos hamiltonianos en los grafos. Un camino hamiltoniano es un recorrido que pasa exactamente una vez por cada nodo del grafo, y un ciclo hamiltoniano es un camino hamiltoniano que además regresa al punto de partida. Para ilustrar estos conceptos, Hamilton diseñó el Juego Icosiano (1857), un rompecabezas comercializado en 1859. En este juego, los participantes debían encontrar un recorrido que visitara 20 vértices de un dodecaedro regular sin repetir nodos y volviendo al punto de inicio. Este problema, aunque planteado en un contexto lúdico, es una de las primeras manifestaciones explícitas del problema del viajante de comercio. [26] Paralelamente, Thomas Penyngton Kirkman, un matemático británico, también estudió estos ciclos en grafos y propuso criterios para su existencia. Sus trabajos ayudaron a comprender mejor la estructura de los grafos en los que es posible trazar un recorrido de este tipo.[27]

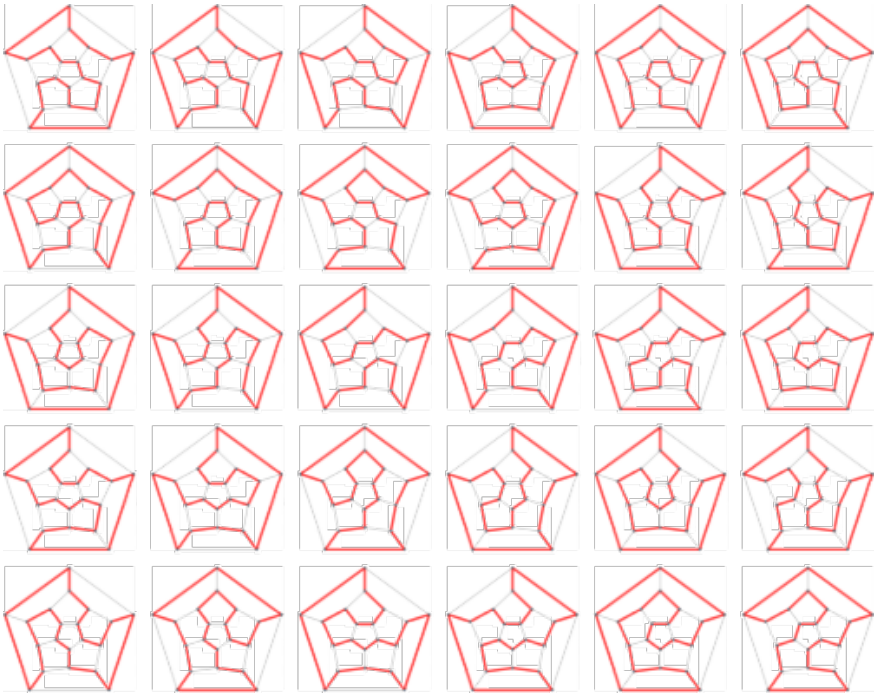

Figura 2. 30 posibles soluciones para el Juego Icosiano. [29]

Si bien tanto Euler como Hamilton investigaron recorridos en grafos, sus enfoques fueron diferentes:

- Euler estudió caminos que visitan todas las aristas de un grafo (caminos Eulerianos), como en el problema de los puentes de Königsberg.
- Hamilton estudió recorridos que visitan todos los nodos una única vez (caminos y ciclos Hamiltonianos), lo que está directamente relacionado con el problema del viajante de comercio.

Mientras que el problema de Euler tiene una solución eficiente basada en criterios matemáticos claros, el problema de Hamilton; y por tanto el TSP, son computacionalmente más complejo y no tiene una solución general sencilla.[28]

2.1.3 Karl Menger y el Problema del Mensajero (1920s): un antecedente directo del TSP.

A comienzos del siglo XX, el matemático austríaco Karl Menger (1902-1985) fue uno de los primeros en estudiar el problema de encontrar una ruta óptima que visitara un conjunto de ciudades con distancias conocidas. En la década de 1920, Menger definió lo que llamó el problema del mensajero, un

problema en el que un cartero debía recorrer varias ubicaciones minimizando la distancia total recorrida. Este problema anticipó muchos de los elementos centrales del TSP, aunque aún no existía una formulación matemática formal ni métodos computacionales para resolverlo.

Menger también formuló una primera versión del algoritmo del vecino más próximo, una heurística en la que el viajante elige siempre la ciudad más cercana como su siguiente destino, método el cual se desarrolla más adelante en este trabajo. Sin embargo, Menger notó que este método no siempre encontraba la solución óptima, sentando así las bases para futuras investigaciones en algoritmos heurísticos y exactos para aproximar el TSP.[30]

Los trabajos de Euler, Hamilton, Kirkman y Menger establecieron el marco teórico sobre el que se desarrollaría formalmente el problema del viajante de comercio en el siglo XX. Los conceptos primarios que emergieron de estos estudios incluyen:

- La representación de rutas mediante grafos.
- La distinción entre caminos Eulerianos y ciclos Hamiltonianos.
- La complejidad de encontrar rutas óptimas en problemas de optimización combinatoria.
- La necesidad de métodos computacionales para abordar problemas con un gran número de nodos.

Estos antecedentes permitieron que, a partir de los años 1930 y 1940, el problema del viajante de comercio fuera formalizado como un problema matemático de optimización, dando inicio a su estudio moderno.

2.1.4 Formalización del TSP (1930-1950). Origen del problema en la planificación de rutas.

El primer gran paso en la formalización del TSP fue dado por el matemático Merrill M. Flood. Flood, quien trabajó en la Universidad de Harvard y tenía un profundo interés en problemas de optimización aplicados, se encontró con una situación práctica que lo llevó a plantear el problema de manera explícita. [31]

El Problema del Viajante de Comercio, tal como se conoce hoy en día, comenzó a tomar su formulación moderna a mediados del siglo XX, cuando los matemáticos e investigadores empezaron a reconocer su importancia en la optimización de rutas y logística. Merrill M. Flood se encontró con el TSP en el contexto de un problema de planificación de rutas en Nueva Jersey en 1937, donde trabajaba en la optimización de los recorridos de autobuses escolares. Su tarea consistía en diseñar un conjunto de rutas eficientes para recoger y dejar a los estudiantes en sus respectivas escuelas, minimizando el tiempo total de recorrido y el costo asociado al transporte. Este problema tenía una analogía directa con el TSP, ya que implicaba la necesidad de visitar un conjunto de ubicaciones; en este caso las escuelas o paradas de autobús, sin repetir ninguna, optimizando la distancia total recorrida.

El desafío que enfrentaba Flood en la optimización de rutas de autobuses escolares era representativo de muchas otras aplicaciones reales del TSP. En esencia, debía responder a la pregunta de cuál es la ruta más corta que permite visitar todas las paradas de autobús exactamente una vez antes de regresar al punto de inicio. Para resolverlo, Flood recurrió a los conceptos matemáticos disponibles en la época, en particular a la teoría de grafos y la programación lineal, campos que estaban en pleno desarrollo. [31] Fue en este proceso que Flood comenzó a discutir el problema con otros matemáticos e investigadores operativos, entre ellos Albert W. Tucker, que con influencia del físico y matemático Hassler Whitney; el cual hizo numerosos estudios sobre la teoría de grafos, fueron los primeros en mencionar una versión primitiva sobre el TSP. [32][33]

2.1.5 El estudio de Dantzig, Fulkerson y Johnson (1954): el primer enfoque computacional.

El siguiente gran avance en la formalización del TSP vino de la mano de tres matemáticos pioneros en la optimización matemática y la programación lineal:

- George B. Dantzig, creador del método del simplex lo cual es un algoritmo de optimización matemática para resolver problemas de programación lineal cuyo objetivo es encontrar el valor óptimo de una función lineal sujeta a un conjunto de restricciones lineales. El método del simplex trabaja sobre un poliedro de soluciones factibles y se desplaza de un vértice a otro siguiendo las aristas de la región factible hasta alcanzar el vértice óptimo. Utiliza una estrategia sistemática para mejorar iterativamente la solución, asegurando que en cada paso la función objetivo mejora o se mantiene constante. [34]

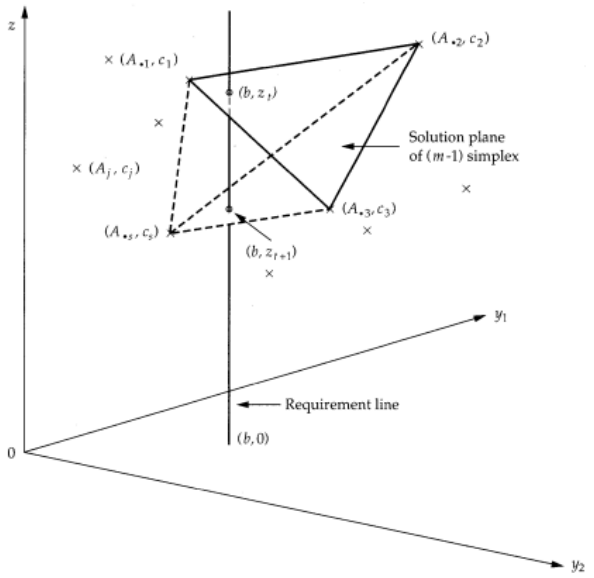

Figura 3. Simplex m-dimensional. [34]

- Ray Fulkerson, experto en teoría de grafos y algoritmos combinatorios.
- Selmer Johnson, especialista en programación lineal entera.

En 1954, estos tres investigadores publicaron el influyente artículo "Solution of a Large-Scale Traveling-Salesman Problem", donde aplicaron técnicas avanzadas de programación lineal entera para resolver el TSP. Hasta ese momento, el TSP era considerado un problema de interés teóricamente hablando, pero sin métodos eficientes para resolverlo en problemas grandes. Dantzig, Fulkerson y Johnson lograron resolver una casuística con 49 ciudades de los Estados Unidos, utilizando herramientas matemáticas innovadoras. Su enfoque consistió en formular el problema como un modelo de programación lineal, utilizar planos de corte para eliminar soluciones no factibles y aplicar técnicas de ramificación y acotación (Branch and Bound) para dividir el problema en partes más pequeñas y manejables. [35]

Figura 4. Matriz del problema de 49 ciudades [35]

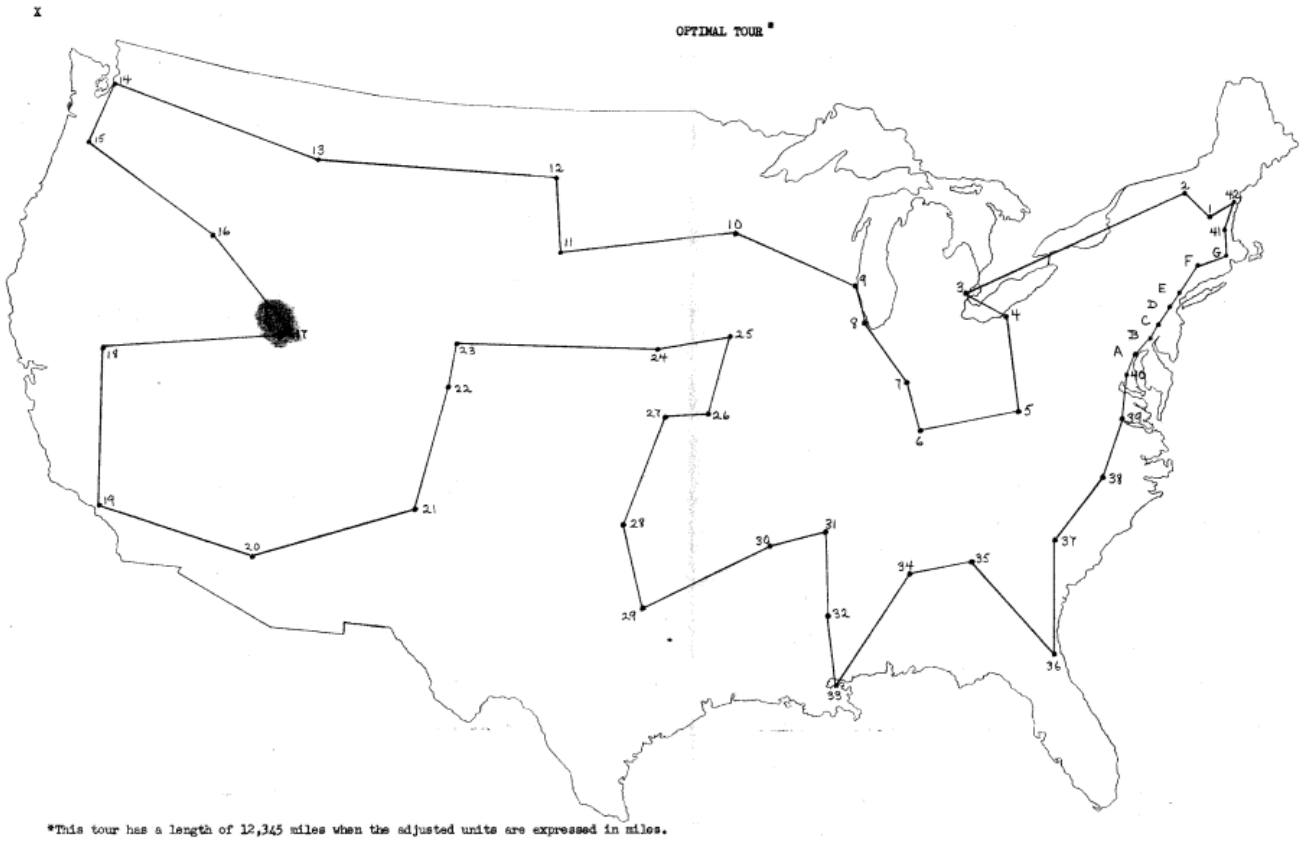

Figura 5. Ruta óptima hallada para matriz de la figura 4. [35]

2.1.6 Planos de Corte

Uno de los aspectos más revolucionarios del estudio de Dantzig, Fulkerson y Johnson fue la introducción de los planos de corte. El problema del TSP, cuando se formula como un problema de asignación; es decir, asegurando que cada ciudad tenga una conexión de entrada y una de salida, a menudo genera soluciones en las que se crean subcircuitos. Estos son conjuntos de ciudades que forman pequeños grupos cerrados en lugar de un único recorrido global. Para corregir esto, propusieron un método en el que, tras encontrar una solución inicial, se añadían nuevas restricciones lineales para eliminar estos subcircuitos. Estas restricciones adicionales, denominadas cortes, reducían progresivamente el espacio de soluciones hasta encontrar un recorrido único óptimo.

El algoritmo sigue un proceso iterativo. En primer lugar, se resuelve la relajación correspondiente del TSP sin ningún tipo de restricción. Una vez se ha hecho esto, se verifica si la solución dispone de subcircuitos. Si resulta no haber subcircuitos, la solución es válida. Si, por el contrario, si los hay, se identifican y se generan restricciones adicionales. Una vez agregadas estas restricciones de corte se vuelve a resolver el problema. Este proceso se repite hasta obtener una solución entera sin subcircuitos. [36]

2.1.7 Método de ramificación y acotación (Branch & Bound)

El método de ramificación y acotación es una estrategia de optimización utilizada para resolver problemas combinatorios complejos, entre ellos el TSP. Su importancia radica en que permite encontrar la solución óptima sin necesidad de evaluar todas las combinaciones posibles, lo cual sería inviable en problemas de gran tamaño. Fue desarrollado a partir de los estudios de Dantzig, Fulkerson y Johnson (1954) [35] y, aunque en su trabajo ya aparecían indicios de esta estrategia, no fue hasta la publicación de Little, Murty, Sweeney y Karel (1963) cuando se formalizó como un método estructurado y aplicable a una gran variedad de problemas de optimización combinatoria. [37]

El principio fundamental del método de ramificación y acotación es bastante intuitivo. Se parte de un problema general, cuya solución directa es difícil de obtener, y se divide en subproblemas más pequeños que sean más manejables. Esto se representa mediante un árbol de decisión, en el que cada nodo es un subproblema con ciertas restricciones adicionales y cada rama representa una posible elección dentro del problema. A medida que se generan estos subproblemas, se va construyendo el árbol, explorando cada rama hasta encontrar una solución factible. Sin embargo, no todas las ramas son igual de prometedoras, por lo que se introducen criterios de poda para descartar aquellas que claramente no pueden conducir a una solución óptima, lo que hace que el proceso sea mucho más eficiente.

Para comprender mejor este mecanismo, imaginemos el caso particular del TSP. Teniendo en cuenta que la solución inicial puede contener subcircuitos; es decir, pequeños ciclos dentro del recorrido que impiden formar un único circuito hamiltoniano. Para corregir esto, el método de ramificación divide el problema en varios subproblemas, cada uno con una restricción adicional que prohíbe ciertos subcircuitos. Cada vez que se detecta un subcircuito, se generan nuevas ramas que restringen progresivamente las soluciones hasta que se obtiene un recorrido válido.

Sin embargo, no basta con ramificar el problema de manera arbitraria. Aquí entra en juego la segunda parte clave del método, que es la acotación. En cada nodo del árbol, se calcula una cota inferior que representa el costo mínimo posible que se podría obtener a partir de esa solución parcial. Si en algún momento una rama del árbol da lugar a una solución con un costo superior a una solución óptima previamente encontrada, esa rama se poda y se descarta, ya que cualquier solución derivada de ella sería inequívocamente peor. Esta estrategia de poda es importante para evitar cálculos innecesarios y hacer que el algoritmo sea eficiente incluso en problemas con muchos nodos.

Una de las razones por las que este método ha sido tan exitoso es su capacidad para reducir drásticamente el número de soluciones a evaluar. En un problema de TSP con muchas ciudades, intentar calcular todas las posibles rutas es prácticamente imposible para conjuntos muy grandes. Sin embargo, gracias a la estrategia de poda, se evita explorar rutas que de antemano se sabe que no pueden ser

óptimas, lo que permite encontrar la mejor solución de forma mucho más eficiente. Es por esto que la ramificación y acotación sigue siendo una de las técnicas más utilizadas en la optimización combinatoria.

Si bien el método de ramificación y acotación puede resolver el TSP de manera exacta, su aplicabilidad en conjuntos extremadamente grandes puede verse limitada por la cantidad de memoria y tiempo de cómputo requeridos. Para hacer frente a este desafío, se ha combinado con otras estrategias como los planos de corte y la relajación lagrangiana [36], logrando resolver problemas con miles de nodos. También se ha incorporado en software especializado, como Concorde TSP Solver, que ha sido utilizado para resolver casos con decenas de miles de ciudades.

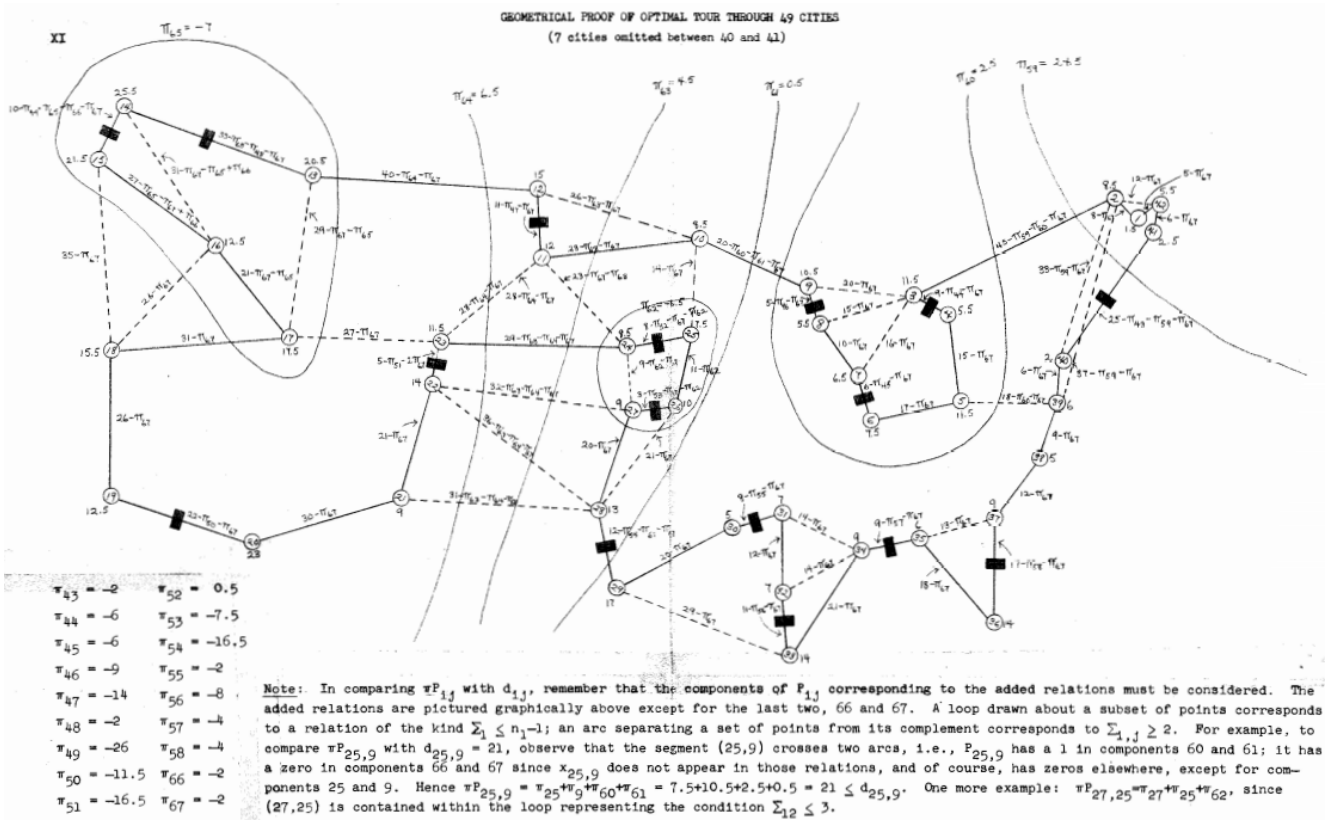

Figura 6. Mapa seccionado con un método precario de Branch & Bound para su resolución. Mismo problema que las figuras 3 y 4. [35]

2.1.8 Origen de la clasificación del problema

Uno de los avances de mayor relevancia en la evolución del TSP se produjo en el campo de la complejidad computacional. A medida que los investigadores desarrollaban métodos para resolver problemas cada vez mayores del TSP durante las décadas de 1950 y 1960, también comenzaba a elaborarse un marco teórico que permitiera clasificar y entender la dificultad inherente de los problemas computacionales. Este marco teórico culminaría en la década de 1970 con la consolidación de las clases P, NP, NP-hard y NP-completo. El desarrollo formal de esta teoría comenzó con el trabajo de Stephen Cook, quien en 1971 publicó su artículo “The complexity of theorem-proving procedures”. [39]

A partir de este trabajo, otros investigadores como Richard Karp profundizaron en esta teoría. En su influyente artículo de 1972 “Reducibility Among Combinatorial Problems” [40], Karp identificó 21 problemas NP-completos, entre ellos el problema del ciclo hamiltoniano. El TSP puede considerarse una generalización del problema del ciclo hamiltoniano. Mientras que este último consiste en determinar si existe un recorrido que visite todos los vértices de un grafo exactamente una vez y regrese al punto de inicio, el TSP añade una función de costes, buscando la ruta de menor peso entre todas las posibles. Esta característica adicional no alivia la dificultad del problema, sino que la mantiene o incluso la agrava desde el punto de vista computacional.

2.1.9 Avances computacionales y algoritmos exactos

Tras la formulación matemática del TSP, el interés por su resolución exacta se convirtió en un desafío prioritario para matemáticos e informáticos. A pesar de que se trataba de un problema con una complejidad exponencial, los avances tanto en algoritmos como en capacidad computacional permitieron abordar problemas cada vez mayores con soluciones óptimas. Esta etapa en la historia del TSP está marcada por la evolución de los algoritmos exactos, diseñados para garantizar la obtención de la mejor solución posible, aunque con elevados costes computacionales.

Con el paso del tiempo, el progreso en los algoritmos exactos fue acompañado por el aumento de la capacidad computacional. Esto permitió afrontar problemas del TSP con un número creciente de ciudades. Sin embargo, fue durante la década de 1990 cuando se alcanzó un punto de inflexión con el desarrollo de herramientas software especializadas. El caso más destacado fue el de Concorde TSP Solver, un software de código altamente optimizado desarrollado por David Applegate, Robert Bixby, Vasek Chvátal y William J. Cook. Concorde implementa una combinación refinada de algoritmos, ramificación y acotación, planos de corte, generación de árboles mínimos y relajaciones lagrangianas, entre otros. Gracias a este enfoque híbrido, Concorde logró resolver problemas de hasta 85.900 nodos en el año 2006, correspondientes a aplicaciones en diseño de circuitos electrónicos, lo que representó un hito mundial en el ámbito de la optimización exacta. [41]

Año	Autores	Nodos
1954	G. Dantzig, R. Fulkerson, S. Johnson	49
1971	M. Held, R.M. Karp	64
1975	P.M. Camerini, L. Fratta, F. Maffioli	67
1977	M. Grötschel	120
1980	H. Crowder and M. W. Padberg	318
1987	M. Padberg and G. Rinaldi	532
1987	M. Grötschel and O. Holland	666
1987	M. Padberg and G. Rinaldi	2392
1994	D. Applegate, R. Bixby, V. Chvátal, W. Cook	7397
1998	D. Applegate, R. Bixby, V. Chvátal, W. Cook	13509
2001	D. Applegate, R. Bixby, V. Chvátal, W. Cook	15112
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook	18512
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, K. Helsgaun	24978
2004	W. Cook, Espinoza and Goycoolea	33810
2006	D. Applegate, R. Bixby, V. Chvátal, W. Cook	85900

Figura 7. TSP resuelto mediante métodos exactos a través de los años. [1]

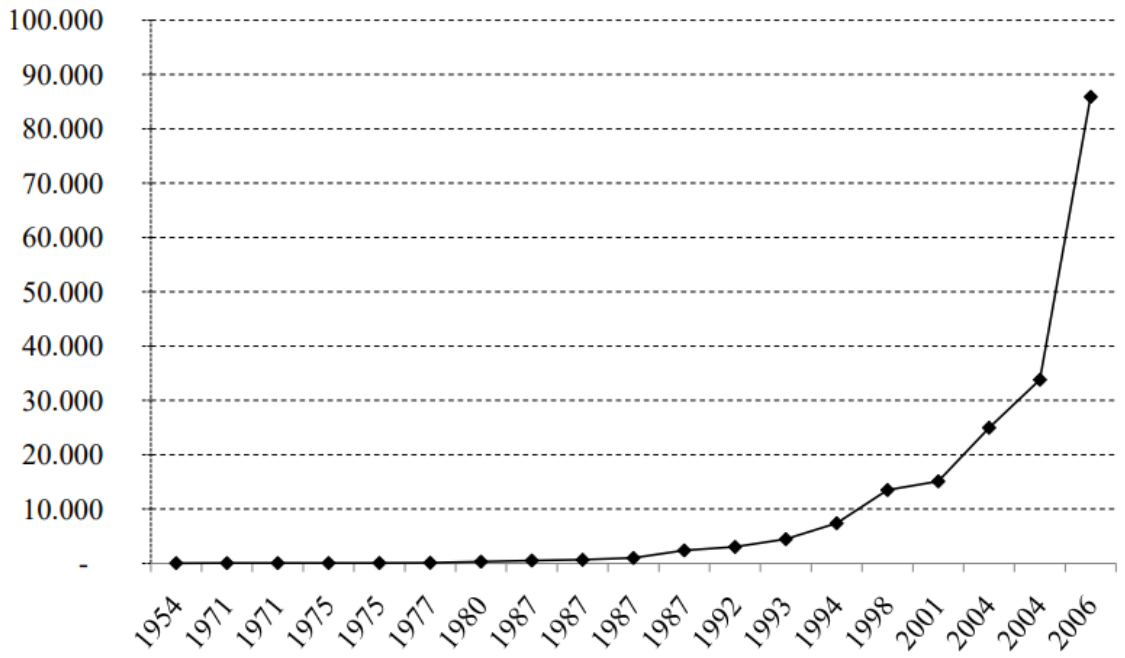

Figura 8. Gráfica exponencial de los avances en resolución de métodos exactos. [1]

2.1.10 Métodos heurísticos, aproximados y metaheurísticos

La aparición de los métodos heurísticos y aproximados para el TSP, surge como respuesta natural a las limitaciones de los métodos exactos, especialmente ante el crecimiento exponencial de la complejidad del problema. A partir de mediados del siglo XX, comenzaron a desarrollarse enfoques prácticos que ofrecían soluciones rápidas, aunque no óptimas.

El primero en proponer una heurística fue Karl Menger en 1931, con el llamado método del vecino más próximo, basado en elegir en cada paso la ciudad más cercana no visitada [30]. Poco después surgieron otras estrategias similares, como los métodos de inserción más barata o métodos basados en árboles de mínima expansión, que construyen soluciones paso a paso en función de ciertos criterios de coste. [42]

Uno de los avances más notables, se dio en 1976 con el algoritmo de Christófides, que ofreció por primera vez una garantía teórica de aproximación, ya que su recorrido no supera el 150 % del óptimo en ningún caso. Esta fue la primera aproximación polinómica con cota conocida y marcó un hito en el diseño de algoritmos aproximados. [2]

A partir de la década de 1980, con el crecimiento de la capacidad computacional, se empezó a aplicar un nuevo enfoque denominado metaheurística. A diferencia de las heurísticas clásicas, estas estrategias incorporan mecanismos de búsqueda más globales, con mayor flexibilidad y capacidad de escape frente a métodos óptimos. Cabe destacar los siguientes métodos metaheurísticos:

- En primer lugar, los algoritmos genéticos. Estos algoritmos se aplican al TSP como una estrategia para encontrar rutas cercanas al óptimo de forma eficiente. En este enfoque, cada posible solución, se representa como un "individuo" o cromosoma, y un conjunto de soluciones forma una población. A través de mecanismos análogos a la selección natural, el cruce de rutas y mutaciones aleatorias, se generan nuevas soluciones en cada generación con el objetivo de mejorar la calidad general de la población. A lo largo de múltiples iteraciones, el algoritmo tiende a converger hacia recorridos de coste reducido. Aunque no garantiza encontrar la solución óptima, su capacidad para explorar amplias regiones del espacio de soluciones lo convierte en una herramienta poderosa para abordar el TSP en problemas de gran tamaño, especialmente

cuando se combina con operadores especializados diseñados para respetar la estructura del problema. [43]

- Cabe destacar el recocido simulado (Simulated Annealing). Este método es una metaheurística probabilística utilizada para resolver problemas de optimización combinatoria como el TSP. Su nombre y funcionamiento están inspirados en el proceso físico de recocido de metales, donde un material se calienta y se enfría gradualmente para alcanzar una configuración de mínima energía. Aplicado al TSP, el algoritmo parte de una solución inicial; es decir, una ruta completa y, en cada iteración, genera una solución vecina mediante una pequeña modificación, como el intercambio de dos ciudades. Si la nueva solución es mejor, se acepta; si es peor, se puede aceptar con cierta probabilidad decreciente que depende de una "temperatura" que se reduce con el tiempo. Este mecanismo permite escapar de óptimos locales en las primeras fases y enfocar la búsqueda hacia las mejores regiones del espacio de soluciones conforme avanza. Aunque no garantiza encontrar la solución óptima, el recocido simulado ha demostrado ser efectivo, simple de implementar y robusto en la práctica para problemas de mediano y gran tamaño del TSP.

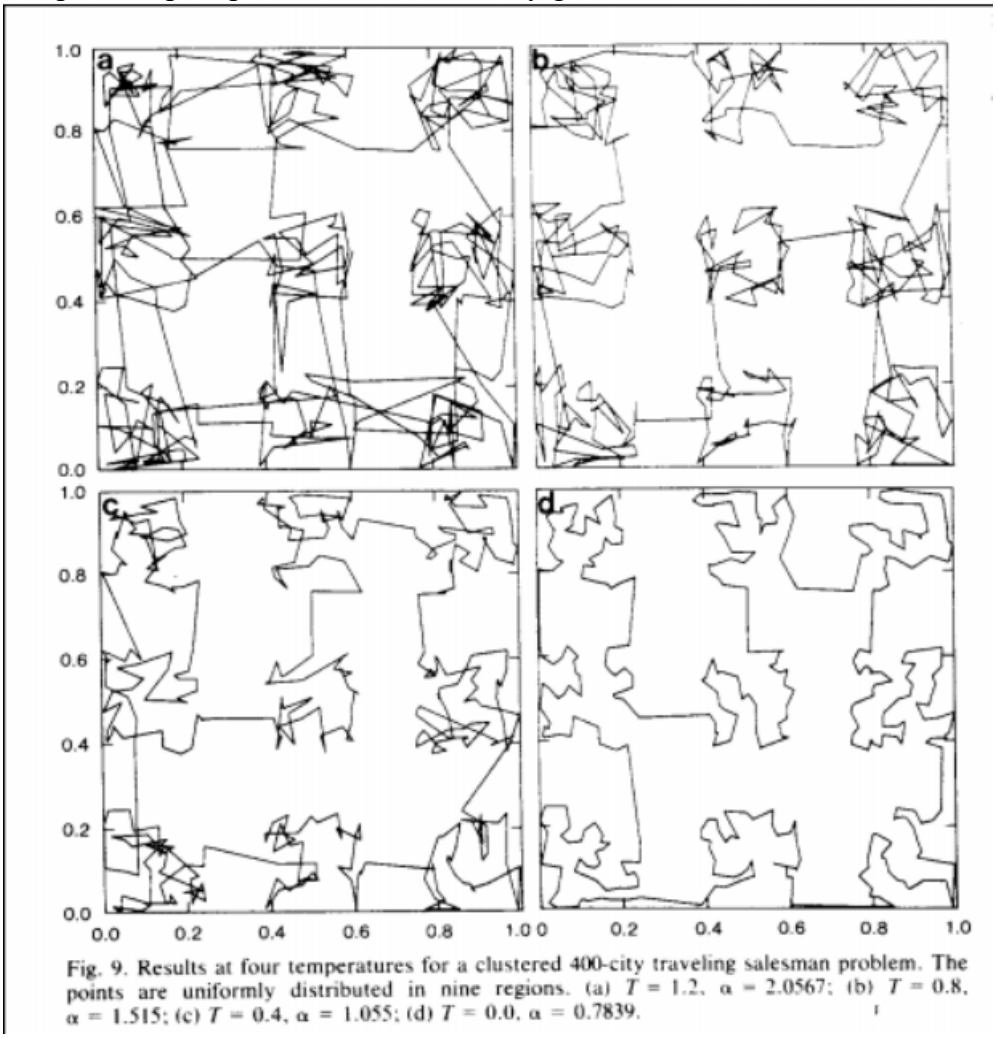

Fig. 9. Results at four temperatures for a clustered 400-city traveling salesman problem. The points are uniformly distributed in nine regions. (a) $T = 1.2$, $\alpha = 2.0567$; (b) $T = 0.8$, $\alpha = 1.515$; (c) $T = 0.4$, $\alpha = 1.055$; (d) $T = 0.0$, $\alpha = 0.7839$.

Figura 9. Ejemplo de 400 ciudades con 4 “temperaturas” distintas en el método del recocido simulado para resolución del TSP. [44]

- Otro método a destacar es el método de la búsqueda del tabú. La búsqueda tabú (Tabu Search) es una metaheurística avanzada utilizada para encontrar soluciones de alta calidad en problemas de optimización como el TSP. A diferencia de las heurísticas clásicas que pueden quedar atrapadas en óptimos locales, la búsqueda tabú introduce una memoria adaptativa que registra los movimientos recientes considerados como “prohibidos” o tabú durante un número determinado de iteraciones. Esta restricción evita que el algoritmo regrese repetidamente a

soluciones ya exploradas, fomentando así una exploración más amplia del espacio de soluciones. A partir de una solución inicial, el algoritmo genera vecinos mediante operaciones como el intercambio de aristas, y selecciona la mejor solución viable, incluso si no mejora inmediatamente el valor de la función objetivo. Gracias a esta estrategia de control de memoria y exploración intensiva, la búsqueda tabú se ha consolidado como una técnica eficaz para obtener soluciones competitivas del TSP. [45]

- Otro método remarcable, es el de la colonia de hormigas. El algoritmo de colonia de hormigas (Ant Colony Optimization, ACO) es una metaheurística bioinspirada que se aplica con éxito al Problema del Viajante de Comercio, imitando el comportamiento colectivo de las hormigas al buscar rutas eficientes entre su nido y las fuentes de alimento. En este enfoque, múltiples agentes artificiales; en este caso las hormigas, construyen recorridos completos por las ciudades, guiándose tanto por la distancia como por la cantidad de feromonas virtuales depositadas en las aristas, que representan la calidad de las soluciones previas. Cuanto mejor es un recorrido, mayor es la cantidad de feromonas depositadas, lo que aumenta su probabilidad de ser elegido por futuras hormigas, promoviendo así la explotación de buenas soluciones. A lo largo de múltiples iteraciones, las feromonas se evaporan parcialmente, lo que permite evitar un estancamiento prematuro y favorece la exploración de nuevas rutas. Este mecanismo de cooperación indirecta y aprendizaje distribuido convierte al ACO en un método potente y adaptable, capaz de encontrar soluciones de alta calidad al TSP, especialmente útil en entornos dinámicos o con restricciones adicionales. [46]

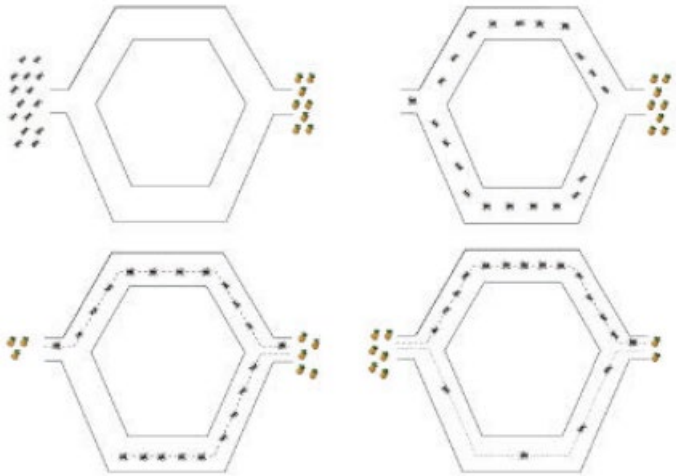

Figura 10. El comportamiento de la colonia termina por obtener el camino más corto entre dos puntos. [46]

2.2 Complejidad del problema.

El Problema del Viajante de Comercio (TSP) se clasifica como $NP-hard$, donde "NP" corresponde a las siglas en inglés de Nondeterministic Polynomial time. Un problema se considera NP si, dada una solución propuesta, es posible verificar su corrección en un tiempo polinómico, es decir, en un tiempo que crece de manera proporcional a una potencia del tamaño de la entrada. Sin embargo, la clasificación NP-hard implica que el problema es, al menos, tan difícil como cualquier otro problema en la clase NP . Formalmente, un problema se denomina NP-hard si la existencia de un algoritmo eficiente para resolverlo implicaría que todos los problemas en NP podrían resolverse de manera eficiente. En otras palabras, si se encontrara un método para resolver un problema NP-hard en tiempo polinómico, esto tendría consecuencias revolucionarias para la teoría de la computación, ya que permitiría resolver fácilmente cualquier problema en NP.

Esta característica convierte al TSP en un problema de gran interés tanto en el ámbito académico como en aplicaciones prácticas. Cualquier avance significativo en la resolución de problemas NP-hard tendría un impacto profundo en la resolución de una amplia gama de problemas computacionales, muchos de los cuales son de menor complejidad y están clasificados dentro de la clase NP. No obstante,

los problemas NP-hard no necesariamente pertenecen a la clase NP , aunque comparten la propiedad de ser extremadamente difíciles de resolver. En particular, la resolución de un problema NP-hard, como el TSP, requiere un tiempo que crece de manera exponencial con el tamaño del problema, lo que lo hace computacionalmente intratable para entradas grandes. [21]

2.2.1 Implicaciones

La clasificación del TSP como un problema $NP - hard$ conlleva dos implicaciones principales:

- La primera es la inexistencia de algoritmos eficientes conocidos. Hasta la fecha, no se conoce ningún algoritmo capaz de resolver el TSP de manera eficiente para problemas arbitrariamente grandes. Esto significa que, aunque es posible resolver el problema para casos pequeños o medianos utilizando métodos exactos, como la programación dinámica o técnicas de ramificación y poda, estos enfoques no escalan bien para ejemplos más grandes debido a su complejidad exponencial.
- En segundo lugar, existe un consenso amplio en la comunidad científica de que es improbable que exista un algoritmo eficiente; es decir, de tiempo polinómico para resolver el TSP. Esta creencia se basa en la hipótesis ampliamente aceptada de que $P \neq NP$, lo que implica que los problemas en la clase P (aquellos que pueden resolverse en tiempo polinómico) son fundamentalmente más simples que los problemas en NP. Si se demostrara que $P = NP$, esto implicaría que todos los problemas en NP, incluidos los $NP - hard$, podrían resolverse de manera eficiente, lo que contradice la intuición y la experiencia acumulada en la teoría de la computación. Por lo tanto, la dificultad intrínseca del TSP y su clasificación como $NP - hard$ lo convierten en un problema de gran relevancia teórica y práctica, ya que su resolución eficiente tendría implicaciones profundas no solo para la optimización combinatoria, sino también para la teoría de la complejidad computacional en su conjunto. [21]

2.2.2 Soluciones según el número de nodos

Cómo previamente se ha introducido, al tratarse de un problema $NP - hard$, el número de soluciones a dicho problema aumenta factorialmente respecto al número de nodos del mismo. El número de soluciones viene definido por la expresión:

$$\frac{(n-1)!}{2}$$

en la que n representa el número de nodos.

La razón de ser de dicha fórmula, se debe a que existen $(n-1)!$ permutaciones posibles de los nodos de un grafo. Al mismo tiempo, esta expresión se divide entre dos ya que cada ruta es posible recorrerla en ambas direcciones.

Asimismo, si disponemos de pocos nodos, se puede alcanzar una solución óptima, mediante el empleo de fuerza bruta; es decir, comprobando la totalidad de soluciones posibles hallando así con exactitud la ruta real más corta. Ejemplificando lo anterior, si se dispusiese de un grafo con únicamente tres nodos; tales cómo A, B y C, tendríamos únicamente una posible solución:

$$A \rightarrow B \rightarrow C \rightarrow A$$

o

$$A \rightarrow C \rightarrow B \rightarrow A$$

dependiendo del sentido de circulación en dicho ciclo.

Al existir únicamente esta casuística, se convierte en trivial el hecho de hallar el peso del camino realmente más corto, mediante la prueba de este único camino como método resolutivo.

Este método resolutivo pasa a ser paulatinamente más ineficiente conforme aumenta el número de nodos. Esto sucede de forma muy rápida como podemos apreciar en la siguiente tabla que relaciona el número de nodos con el número de rutas asociadas a cada casuística.

NODOS	SOLUCIONES
3	2
5	12
7	360
9	20160
11	1814400
13	239500800
15	$4{,}35 \times 10^{10}$
17	$1{,}05 \times 10^{10}$
20	$6{,}08 \times 10^{16}$
40	$1{,}02 \times 10^{46}$
60	$6{,}93 \times 10^{79}$

Tabla 1. Elaboración propia curso 2025. Relación entre el número de nodos y el número de circuitos hamiltonianos solución de ese problema.

2.2.3 Computación temporal y notación asintótica

En análisis de algoritmos, es de interés medir la eficiencia temporal; es decir, cuánto tiempo tarda un algoritmo en ejecutarse en función del tamaño de dicho problema. Para ello, se utiliza la notación asintótica, que permite describir cómo se comporta el tiempo de ejecución cuando el tamaño de la entrada n tiende a infinito. Esto proporcionará una herramienta útil para clasificar los distintos métodos que se tratan en este trabajo.

La notación O grande, $(O(f(n)))$ proporciona una cota superior asintótica, lo cual indica que, para valores suficientemente grandes de n, el tiempo de ejecución del algoritmo no crece más rápido que una función $f(n)$ multiplicada por una constante.

Definición 2.2.3.1. [2] Se define que una función $T(n)$ pertenece a $O(f(n))$ si existen dos constantes positivas c y n_0 tales que para todo $n \geq n_0$ se cumple que:

$$T(n) \leq c \times f(n)$$

Esto significa que el tiempo de ejecución del algoritmo está acotado superiormente por $f(n)$, salvo por una constante multiplicativa. Para valores grandes de n, el crecimiento de $T(n)$ no supera la función $f(n)$, permitiendo comparar algoritmos en función de su eficiencia. Los principales tipos de complejidad $O(f(n))$, que aparecen nombrados en diversos puntos de este trabajo, son los siguientes [47]:

1) En primer lugar, existe la complejidad constante, con notación $O(1)$. Su tiempo de ejecución no depende del tamaño del problema. Independientemente de cuanto crezca n, el número de operaciones realizadas es siempre el mismo.
2) Complejidad logarítmica. Un algoritmo es $O(\log n)$ si, en cada paso, reduce el tamaño del problema a una fracción de su tamaño original.
3) Complejidad lineal $O(n)$. Un algoritmo es $O(n)$ si el tiempo de ejecución crece proporcionalmente al tamaño de la entrada.
4) Complejidad cuadrática $O(n^2)$. Un algoritmo es $O(n^2)$ si el tiempo de ejecución es proporcional al cuadrado del tamaño de la entrada. Suele ocurrir en algoritmos con bucles animados.
5) Complejidad cúbica. Un algoritmo es $O(n^3)$ si el tiempo de ejecución es proporcional al cubo del tamaño de la entrada.
6) Complejidad exponencial $O(2^n)$. Un algoritmo es $O(2^n)$ si el tiempo de ejecución se duplica con cada incremento en la entrada.

7) Complejidad factorial $O(n!)$. Un algoritmo es $O(n!)$ si el tiempo de ejecución crece de forma factorial con el tamaño de la entrada, lo que resulta completamente intratable para valores grandes de n.

MÉTODO	TIPO	COMPLEJIDAD TEMPORAL	DESCRIPCIÓN
Fuerza Bruta	Exacto	$O(n!)$	Evalúa todas las soluciones posibles
Bellman-Held-Karp	Exacto	$O(2^n \times n^2)$	Más eficiente que fuerza bruta pero aún exponencial
1-árbol	Aproximado	$O(n^3)$	Proporciona cota inferior, basada en MST
Christófides	Aproximado	$O(n^3)$	Garantiza solución $1{,}5 - aproximado$
Branch & Bound	Aproximado	$O(n!)$ en el peor caso, $O(c^n)$ en promedio	Explora y descarta soluciones no viables
Inserción más cercana	Heurístico	$O(n^2)$	Solución añadiendo nodo más cercano
Árbol	Heurístico	$O(n^2 \log(n))$	Usa árboles generadores mínimos

Tabla 2. Elaboración propia curso 2025. Comparación de métodos con sus respectivas complejidades temporales.[47]

3 DESARROLLO DEL TFG

3.1 Introducción

El Problema del Viajante de Comercio (TSP), es uno de los problemas más relevantes en la teoría de la optimización combinatoria y ha sido ampliamente estudiado debido a su complejidad computacional y sus aplicaciones prácticas como en logística, telecomunicaciones y bioinformática [11]. Su formulación matemática consiste en encontrar un ciclo hamiltoniano de costo mínimo en un grafo completo ponderado, donde los pesos representan las distancias o costos entre los nodos.

Dado que el TSP pertenece a la clase de problemas NP-hard, no se conoce un algoritmo en tiempo polinomial que garantice la obtención de la solución óptima en todos los casos. De hecho, determinar si existe un ciclo hamiltoniano en un grafo arbitrario ya es un problema NP-completo. Esto implica que, salvo que P = NP, cualquier algoritmo exacto que resuelva el TSP requerirá en el peor de los casos un tiempo exponencial en función del número de nodos.

Para abordar esta dificultad, se han desarrollado distintos enfoques de resolución, los cuales pueden clasificarse en tres grandes categorías según su precisión y eficiencia computacional.

Métodos exactos: garantizan la obtención de la solución óptima, pero su complejidad suele hacerlos inviables para problemas de gran tamaño. Entre estos se encuentran la fuerza bruta, la programación dinámica de Held-Karp [3] y la programación entera mixta con técnicas de ramificación y acotación.

Métodos aproximados: se diseñan para obtener soluciones que se encuentran dentro de un margen de error conocido con respecto al óptimo. Un ejemplo destacado es el algoritmo de Christófides, que proporciona una solución cuyo costo no supera en más de un 50% el valor óptimo en el peor de los casos [12].

Métodos heurísticos: ofrecen soluciones subóptimas en tiempos de ejecución significativamente menores, lo que los hace adecuados para aplicaciones en tiempo real. Algunos de los enfoques más utilizados incluyen el algoritmo del vecino más cercano, la heurística de inserción y los métodos basados en árboles de expansión mínima [3].

El objetivo de este capítulo es analizar estos enfoques en detalle, comparando su rendimiento desde el punto de vista teórico y empírico. Se evaluará su eficiencia computacional, su aplicabilidad en distintos contextos y su desempeño en situaciones de diferentes tamaños. Además, se discutirán los avances recientes en el campo de la optimización heurística y metaheurística.

El estudio del TSP no solo es de interés teórico, sino que también tiene implicaciones prácticas significativas. Desde su formulación en el siglo XIX, ha sido utilizado para modelar problemas de enrutamiento en la industria del transporte, la optimización de redes de distribución y la secuenciación de tareas en manufactura [1]. Su resolución eficiente continúa siendo un desafío abierto en la investigación en optimización y ciencias de la computación.

3.1.1 Formulación matemática TSP y su relación con la teoría de grafos.

Definición 3.1.1.1. Sea un grafo completo $G = (V, E)$, donde $V = \{1,2, \dots, n\}$ representan el número de nodos o ciudades y $E \subseteq V \times V$ es el conjunto de aristas o conexiones entre ciudades, se asocia a cada arista $(i, j) \in E$ un coste $c_{ij} \in \mathbb{R}^+$, que representa; por ejemplo, la distancia, tiempo o coste económico de viajar de la ciudad i a la ciudad j.

El objetivo del TSP es encontrar un ciclo hamiltoniano $C \subset E$; es decir, un recorrido cerrado que pase exactamente una vez por cada vértice de V y que minimice la suma total de los costes asociados a las aristas del ciclo. La formulación matemática clásica del TSP es la siguiente:

- Minimizar

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

- Sujeto a

$$\sum_{j=1, j\neq i}^{n} x_{ij} = 1 \qquad \forall i \in V$$

$$\sum_{i=1, i\neq j}^{n} x_{ij} = 1 \qquad \forall j \in V$$

$$x_{ij} \in \{0,1\} \quad \forall i, j \in V, i \neq j$$

Las dos primeras restricciones garantizan que cada nodo sea visitado exactamente una vez, y que de cada nodo salga y entre exactamente una arista. Sin embargo, estas condiciones no impiden la existencia de subciclos, por lo que es necesario añadir restricciones adicionales para garantizar que la solución encontrada forme un único ciclo que conecte todos los nodos.

Una forma habitual de evitar subciclos es mediante las condiciones de Miller–Tucker–Zemlin (MTZ), que introducen variables adicionales $u_i \in \mathbb{Z}, i = 2, \dots, n$ y que definen las siguientes restricciones:

$$u_i - u_j + n x_{ij} \leq n - 1, \qquad \forall i \neq j, \qquad i, j \in \{2, \dots, n\}$$

Estas restricciones aseguran que el recorrido tenga una única componente conexa y, por tanto, sea un ciclo hamiltoniano válido sin subciclos. El nodo u_1 se suele fijar a 0 como punto de partida del recorrido. Esta formulación es mucho más eficiente computacionalmente que las formulaciones que requieren generar restricciones para subconjunto propio de nodos del grafo.

Desde la perspectiva de la teoría de grafos, el TSP se relaciona directamente con el problema del ciclo hamiltoniano, el cual consiste en determinar si existe un ciclo en un grafo que visite exactamente una vez todos sus vértices. El TSP puede considerarse una extensión optimizada de este problema, en la que se busca el ciclo hamiltoniano de menor coste. [2]

3.2 Métodos exactos

Los métodos exactos para resolver el Problema del Viajante de Comercio, son algoritmos diseñados para encontrar la solución óptima garantizada. Estos métodos, aunque computacionalmente costosos, son de gran interés en contextos donde la precisión es prioritaria. Su principal desventaja radica en que su complejidad temporal crece exponencialmente con el número de nodos, lo que los hace ineficientes para problemas grandes. Entre los más destacados se encuentran la fuerza bruta, la programación dinámica (Bellman-Held-Karp) y el algoritmo de Branch & Bound previamente expuesto. A pesar de sus limitaciones, estos enfoques han sido importantes en la obtención de cotas que mejoran los algoritmos aproximados y heurísticos.

3.2.1 Fuerza bruta

El enfoque por fuerza bruta, evalúa todas las posibles secuencias de recorrido y selecciona la de menor costo.

El procedimiento que sigue es el siguiente:

1) Se establece el número total de posibles permutaciones para un total de n ciudades.

$$(n-1)! = (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

2) Se continúa evaluando el costo de cada ruta. Para cada una de las permutaciones π se realiza este proceso.

$$C(\pi) = \sum_{i=1}^{n-1} d\left(v_{\pi(i)}, v_{\pi(i+1)}\right) + d(v_{\pi(n)}, v_{\pi(1)})$$

3) Se procede a la selección de la ruta más óptima, en la que se almacena la permutación π^* que ostente un menor costo.

$$\pi^* = \arg \min_{\pi \in S_{n-1}} C(\pi)$$

Nota 3.2.1.1. Las permutaciones son formas de ordenar un conjunto de elementos. En otras palabras, una permutación es una disposición distinta de los elementos de un conjunto, donde el orden sí importa.

Observación 3.2.1.2. En cuanto a la complejidad computacional, se resalta que el algoritmo explora $(n-1)!$ posibles soluciones con un costo supuesto de $O(n)$ por cada permutación. De esta forma se puede comprobar que la complejidad total real es:

$$O(n \cdot (n-1)!) \approx O(n!)$$

Este crecimiento factorial deriva en el hecho de que este método se hace impracticable para valores grandes de n. Por ejemplo:

Para $n = 50 \rightarrow 49!$ es del orden de 10^{62}, lo cual supera la capacidad computacional actual. [6]

Ejemplo 3.2.1.3. Se considera un grafo de n=4 ciudades con la siguiente matriz de distancias:

$$D = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$$

Las diversas posibles permutaciones teniendo en cuenta que se ha fijado la ciudad v_1 cómo nodo inicial son las siguientes:

1. $(1,2,3,4), coste\ 10 + 35 + 30 + 20 = 95$
2. $(1,2,4,3), coste\ 10 + 25 + 30 + 15 = 80$
3. $(1,3,2,4), coste\ 15 + 35 + 25 + 20 = 95$
4. $(1,3,4,2), coste\ 15 + 30 + 25 + 10 = 80$
5. $(1,4,2,3), coste\ 20 + 25 + 35 + 15 = 95$
6. $(1,4,3,2), coste\ 20 + 30 + 35 + 10 = 95$

En este caso existen dos soluciones óptimas que son $(1,2,4,3)$ y $(1,3,4,2)$. Ambas tienen un coste final total de 80.

La principal ventaja en el empleo de la fuerza bruta es que con certeza se va hallar la verdadera solución óptima. Este método resulta útil para ejemplos pequeños cómo el expuesto en **3.2.1.2.** En contrapunto, para grafos reales con $n > 10$ es preciso emplear una serie de métodos de mayor eficiencia como la programación dinámica con el algoritmo de Belhman-Held-Karp o ciertos algoritmos aproximados o heurísticos que posteriormente serán tratados en este trabajo.

3.2.2 Algoritmo de Belhman-Held-Karp.

Primero, se ha de reconocer lo siguiente.

Proposición 3.2.2.1. [8] El ciclo Hamiltoniano más corto no depende de la elección del vértice de inicio.

Esto es evidente porque el ciclo hamiltoniano tiene simetría cíclica, lo que implica que cambiar el vértice inicial no cambia el orden de los demás vértices en el ciclo. Por lo tanto, podemos comenzar la construcción desde x_1.

Definición 3.2.2.2. Sea $S \subset V\setminus\{x_1\} = \{x_2, x_3, \dots, x_n\}$ un subconjunto de tamaño s en el que $(1 \leq s \leq n-1)$. Para cada vértice $x_i \in S$, definimos la función de coste $cost(x_i, S)$ como la longitud del camino más corto desde $\boldsymbol{x}_1$ hasta x_i, visitando cada vértice en S exactamente una única vez. De una forma más precisa, se implementa con la siguiente fórmula recursiva:

$$cost(x_i, S) = \min_{xj} \{cost(x_j, S\setminus\{x_i\}) + D_{ji}\}$$

En la cual, $x_j \in S\setminus\{x_i\}$. En el caso de que S tuviese un tamaño igual a 1, se definiría cómo:

$$cost(x_i, S) = D_{1i}$$

Al utilizar esta fórmula recursiva, se puede construir la función de costes de una forma progresiva para subconjuntos S de tamaños comprendidos entre 1 y $n-1$. Gracias a este carácter recursivo, se puede deducir que cada uno de los pasos hallados, es la base para el siguiente paso a tener en cuenta. La implementación de este algoritmo en lenguaje de programación recibe el nombre de "Dynamic Programming" o Programación dinámica en castellano.

Al alcanzarse un subconjunto S de tamaño n-1; es decir, $S = V\setminus\{x_1\}$, el último paso a realizar sería el siguiente:

$$Menor\ ciclo\ hamiltoniano = \min_{x_i}\{cost(x_i, S) + D_{i1}\} \equiv \min_{x_i}\{cost(x_i, V\setminus\{x_1\}) + D_{i1}\}$$

$$\text{Con } x_i \in S \equiv V\setminus\{x_i\}$$

Observación 3.2.2.3. [8] (Complejidad temporal del algoritmo). La complejidad temporal asociada a este algoritmo es: $O(n^2 \times 2^n)$.

Cómo prueba de esto, se puede ver que este algoritmo es considerado un subconjunto 2^{n-1} de $V\setminus\{x_1\}$. Para cada uno de los subconjuntos, se calcula la función de costo para todos sus elementos, para los cuales no hay más que n términos. Para cada una de las evaluaciones realizadas de la función de coste, se consideran como máximo n valores basados en los resultados previamente hallados.

Observación 3.2.2.4.[8] En comparación con el método por fuerza bruta, en el cual el nivel de complejidad viene dado por $O(n!)$. El nivel de complejidad de este algoritmo es significativamente menor. Por ejemplo, para una casuística en la que $n = 100$:

$$O(n!) = O(100!) = O(9.33 \times 10^{157})$$

$$O(n^2 \times 2^n) = \mathrm{O}(100^2 \times 2^{100}) = O(1.27 \times 10^{34})$$

Este resultado es aproximadamente 7.25×10^{123} veces más rápido.

Cabe tener en cuenta que el tiempo de ejecución con el empleo de este algoritmo crece igualmente de manera exponencial conforme crece el número n de ciudades. Aunque este algoritmo mejora significativamente respecto al método de fuerza bruta $O(n!)$, sigue siendo computacionalmente complejo para valores grandes de n.

En la práctica, este algoritmo es aplicable a problemas de tamaño moderado. Por ejemplo, se ha demostrado que es capaz de resolver problemas del viajante de hasta 100 nodos en ordenadores personales en un tiempo razonable, gracias a ciertas modificaciones en el algoritmo original.[2]

Para problemas más grandes, el tiempo de computación aumenta considerablemente, y se requieren técnicas más avanzadas o heurísticas para obtener soluciones aproximadas en tiempos razonables.

3.3 Árbol de expansión mínima (MST)

Antes de entrar en la explicación de diferentes métodos aproximados, se ha introducido el concepto de árbol de mínima expansión debido a su aplicabilidad

Definición 3.3.0.1. Un árbol de expansión mínima, nombrado también cómo MST por sus siglas en inglés (Minimum Spanning Tree), es un subgrafo de un grafo conexo. Es un árbol, lo que quiere decir que se trata de un subgrafo acíclico que conecta todos los vértices del grafo. Entre todos los árboles de expansión posibles, el MST es aquel cuya suma de pesos de las aristas es mínima.

Sea $G = (V, E, w)$ un grafo no dirigido, conexo y ponderado, donde V es el conjunto de vértices y E el conjunto de aristas con una función de peso $w: E \to \mathbb{R}^+$ que asigna un valor positivo a cada arista. Un MST $T \subseteq E$ es un subconjunto de E que conecta todos los vértices en V sin formar ciclos y minimiza la siguiente función objetivo:

$$w(T) = \sum_{e \in T} w(e), \quad donde\ T \subseteq E\ y\ |T| = |V| - 1$$

Teorema 3.3.0.2. Un árbol de expansión T en G es mínimo únicamente si satisface la siguiente propiedad: para toda arista $e \notin T$, si se agrega e a T, se forma un ciclo C, y si existe otra arista $f \in C \backslash \{e\}$ tal que $w(f) > w(e)$, entonces T no es mínimo y puede ser mejorado reemplazando f por e.

Algoritmo 3.3.0.2. Algoritmo de Prim. Este algoritmo construye un MST de manera *greedy*; es decir, añadiendo en cada paso la arista de menor peso que conecta un vértice del árbol en construcción con un vértice fuera de él.

El algoritmo de Prim define el siguiente procedimiento:

- Se inicializa un conjunto de vértices $S = \{v_0\}$, donde v_0 es un vértice arbitrario.
- Para cada vértice $u \notin S$, asignar un valor $g(u) = \infty$, indicando la distancia mínima a S. De existir una arista (v_0, u), se actualiza, y se identifica que $g(u) = w(v_0, u)$.
- Mientras $S \neq V$, se selecciona un vértice $u \notin S$ al cual le corresponda el menor valor de $g(u)$. Dicho valor del vértice u se agrega a S. Posteriormente, se actualizan los valores de $g(v)$ para cada vecino $v \notin S$, si la arista (u, v) tiene menor peso que el valor almacenado en $g(v)$.
- Por último, se lleva a cabo la repetición de este paso hasta incluir la totalidad de los vértices.

Teorema 3.3.0.3. El algoritmo de Prim determina con una complejidad $O(|V|^2)$ un árbol de expansión mínima T para (G, w).

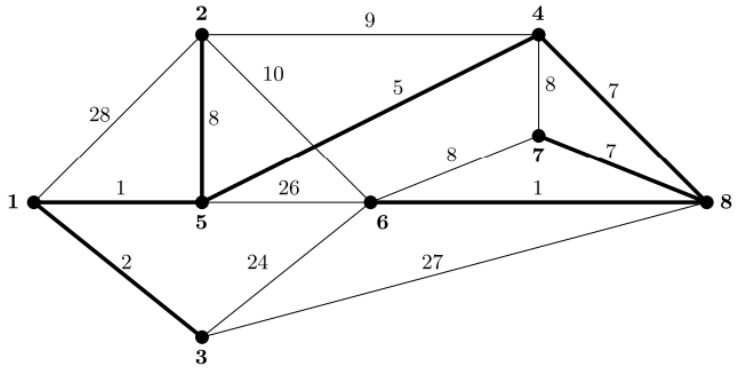

Figura 11. [2] Ejemplo del MST hallado mediante el algoritmo de Prim.

Algoritmo 3.3.0.4. [2] Algoritmo de Kruskal. [2] Sea $G = (V, E)$ un grafo conexo, con $V = \{1, \dots, n\}$, y la función de coste $w: E \to \mathbb{R}$. Las aristas de G se ordenan en función de su peso; tal que, $E = \{e_1, \dots, e_m\}$ con $w(e_1) \leq \cdots \leq w(e_m)$.

El algoritmo de Kruskal; al igual que el método de Prim, también es un método *greedy*, pero siguiendo un enfoque diferente. En este caso ordena las aristas por peso y las agrega al MST en el caso de que no formen ciclos.

El procedimiento que se sigue en la implementación de este algoritmo es el siguiente:

- En primer lugar, se ordenan las aristas en orden creciente de peso.
- Se inicializa un conjunto $T = \emptyset$ en el cual se inicializa el conjunto de aristas vacías
- Para cada arista $e_k = (u_k, v_k)$ siguiendo el orden creciente de los pesos de las aristas; si esta no forma un ciclo T, se añade dicha arista e_k al ciclo T.
- Se establece como condición de parada que si $|T| = |V| - 1$ el proceso ha de terminar.

Se puede de esta forma establecer que a la entrada del algoritmo se dispone de un grafo conexo y ponderado $G = (V, E)$ con pesos $w: E \to \mathbb{R}$; y a la salida se dispone de un MST $T \subseteq E$ con $|V| - 1$ aristas.

La complejidad computacional total de este algoritmo es $O(|E| \log |E|)$ ya que ordenar las aristas toma $O(|E| \log |E|)$ y la unión de componentes se realiza con conjuntos disjuntos con unión por rango y compresión de camino, logrando una complejidad amortizada de $O(|E| \alpha(|E|))$ donde α es la inversa de la función de Ackermann [13], la cual crece extremadamente lento por lo que se puede despreciar.

Algoritmo 3.3.0.5. [14] Algoritmo de Borůvka. Se opera dividiendo el problema en varias componentes conexas, donde en cada iteración se selecciona la arista que aporte un menor coste para cada componente y se fusionan las componentes hasta formar un único MST.

El algoritmo de Borůvka es uno de los primeros algoritmos diseñados para encontrar un MST y fue propuesto por Otakar Borůvka en 1926 [14] para optimizar redes eléctricas.

Se puede asumir que a la entrada del algoritmo se dispone de un grafo conexo y ponderado $G = (V, E)$ con pesos $w: E \to \mathbb{R}$ y a la salida se tiene un MST $|V| - 1$ aristas.

El procedimiento formal que se utiliza para utilizar este algoritmo es el siguiente:

- En primer lugar, se define cada vértice como una componente conexa separada dónde T$\leftarrow\emptyset$ inicializando el conjunto vacío de aristas.
- A continuación, se procede a la fase de iteración. En cada iteración, cada componente selecciona su arista de menor peso hacia otra componente distinta.
- Se agregan todas las aristas seleccionadas al MST, fusionando los componentes.
- Se repite hasta que todas las componentes se hayan fusionado en un solo MST.

Dado que el número de componentes al menos se reduce a la mitad en cada iteración, el algoritmo se ejecuta en $O(\log |V|)$ iteraciones.

Observación 3.3.0.6. El algoritmo de Prim es más eficiente en grafos densos, pues trabaja con vértices. El algoritmo de Kruskal es útil ante todo en grafos dispersos, ya que solamente trabaja con aristas. El algoritmo de Borůvka es útil en arquitecturas paralelas ya que permite agregar múltiples aristas en una única iteración.

Algoritmo	Estrategia	Estructura utilizada	Complejidad en grafos densos	Complejidad en grafos dispersos	Ventaja principal	Desventaja principal
Prim	Expansión progresiva	Cola de prioridad	$O(V^2)$ con lista de adyacencia	$O(ElogV)$	Eficiente en grafos densos	Puede ser más lento en grafos dispersos
Kruskal	Selección de aristas	Conjuntos disjuntos	$O(ElogE)$ con ordenación de aristas	$O(ElogV)$	Eficiente en grafos dispersos	Requiere ordenar las aristas
Borůvka	Método de división	Componentes conexas	$O(ElogV)$ fusionando componentes en cada iteración	$O(ElogV)$	Buena implementación paralela	Menos utilizado en la práctica

Tabla 3. Elaboración propia curso 2025. Resumen comparativo de los métodos para generar MST. [2]

3.4 Métodos aproximados

Los métodos aproximados para la resolución del Problema del Viajante de Comercio (TSP) son algoritmos que no siempre encuentran la solución óptima, pero garantizan que el costo de la solución obtenida se encuentre dentro de un factor constante del costo óptimo. Se hallará entonces, con la implementación de este tipo de métodos una solución subóptima dentro de ese factor de aproximación.

Definición 3.4.0.1. Sea P un problema de optimización de minimización definido sobre un conjunto de problemas I, donde para cada problema $x \in I$ existe un conjunto de soluciones factibles $S(x)$ y una función de costo $f: S(x) \to \mathbb{R}^+$.

Un algoritmo de aproximación A genera una solución $A(x) \in S(x)$ tal que su costo $f(A(x))$ cumple la siguiente desigualdad en el peor de los casos:

$$OPT(X) \leq f(A(x)) \leq \alpha \times OPT(X), \quad \forall x \in I$$

Donde:

- $OPT(x)$ se refiere al valor de la solución óptima para un problema x.
- $\alpha \geq 1$ es el factor de aproximación que representa la cota superior de la desviación respecto al resultado óptimo.

Esto implica que un algoritmo de aproximación encuentra soluciones cuyo valor es; como máximo, un $\alpha \cdot factor\ del\ óptimo$.

Ejemplo 3.3.0.2. Centrándose ya en el TSP plenamente, un algoritmo de aproximación genera un ciclo hamiltoniano π' tal que:

$$C(\pi') \leq \alpha \cdot C(\pi^*)$$

Donde:

- $C(\pi^*)$ es el coste del recorrido óptimo.
- $C(\pi')$ es el coste de la solución obtenida por el algoritmo aproximado.
- α es el factor de aproximación del algoritmo.

3.4.1 Cálculo de una cota inferior.

El proceso para hallar una cota inferior para el TSP se centrará en este caso en la utilización del método del $1-\acute{A}rbol$.

El método del $1-\acute{A}rbol$; más comúnmente referenciado por su nombre en inglés $1-tree$, es una técnica utilizada para encontrar una cota inferior del coste óptimo del Problema del Viajante. La idea principal de este método consiste en construir una estructura similar a un árbol de expansión mínima [10], pero con la principal particularidad de que un nodo específico previamente seleccionado, llamado nodo raíz s, tiene $grado$ 2 en la solución final; es decir, que tiene dos aristas que lo unen a otros nodos.

Definición 3.4.1.1. Sea $G=(V,E)$ un grafo completo con un conjunto de vértices V y un conjunto de aristas E ponderadas por una función de costes $d: E \rightarrow \mathbb{R}^{+}$ cuyos costes vienen señalados por una matriz $W=(w_{ij}), i,j=1,\dots,n$ en el caso de un problema TSP. Un $1-\acute{A}rbol$ es una estructura conformada por:

- Un árbol de expansión mínima sobre los vértices $V\backslash\{s\}$.
- Dos aristas adicionales que conectan el nodo raíz s con otros dos nodos de árbol de expansión mínima.

Dado que todo ciclo hamiltoniano en un grafo de n vértices contiene exactamente n aristas, el $1-\acute{A}rbol$ representa una relajación del TSP, permitiendo obtener una cota inferior del coste óptimo del problema.

Para llevar a cabo este algoritmo se siguen los siguientes pasos:

1. Se escoge un vértice $s \in \{1,\dots,n\}$ del grafo G y se elimina.
2. Se seleccionan las dos aristas del conjunto de aristas E que tengan un menor coste w pero que a su vez incidan sobre el vértice s previamente seleccionado.
3. Se toma el grafo $G-\{s\}$ y se procede a calcular un árbol de expansión de peso mínimo Q empleando alguno de los métodos nombrados en 3.4.1.
4. Se calcula el coste total de dicho árbol $w(Q)$ y se le suma el coste de las otras dos aristas incidentes sobre el vértice s. Esto garantiza que el nodo s tenga grado 2, lo cual hace que se cumpla la estructura del $1-\acute{a}rbol$.

El coste total del $1-\acute{a}rbol$ vendrá dado por la siguiente expresión:

$$w(1-tree)=w(MST)+d(s,u)+d(s,v)$$

Donde u y v son los dos vértices más cercanos a s para conectar con él como se menciona previamente en el paso 4.

El valor del $1-\acute{a}rbol$ proporciona una cota inferior al coste real del TSP, siendo ésta siempre menor o igual a dicho coste:

$$L(1-tree) \leq w(TSP)$$

Observación 3.4.1.2. Dado que un ciclo hamiltoniano óptimo es también un $1-\acute{a}rbol$, su peso será al menos el del $1-\acute{a}rbol\ mínimo$. La calidad de esta cota depende de la selección del nodo s, pues diferentes elecciones del nodo de origen, pueden generar diferentes valores de cota inferior.

Ejemplo 3.4.1.3. Se supone un conjunto de 4 ciudades al cual le corresponde la siguiente matriz de costes:

$$D=\begin{bmatrix}0 & 2 & 9 & 4\\ 2 & 0 & 8 & 4\\ 9 & 8 & 0 & 3\\ 4 & 4 & 3 & 0\end{bmatrix}$$

1. Se tomará $s = 1$ cómo nodo raíz.
2. Se procede a la construcción del MST sobre el subgrafo restante $V\backslash\{s\}$. En este caso, el MST correspondiente mediante el método de Kruskcal [10] sobre $\{2,3,4\}$ tiene un coste de 7.
3. Se seleccionan las dos aristas de menor coste incidentes desde 1. Éstas son $d(1,2) = 2$ y $d(1,4) = 4$.
4. Se suman estos dos valores al coste del árbol de mínima expansión.

$$L(1-tree) = 7 + 2 + 4 = 11$$

Mediante método de fuerza bruta, se ha identificado que la solución óptima corresponde a la ruta $1 \to 2 \to 4 \to 3 \to 1$ con un coste total $w(TSP) = 13$.

De este modo se puede verificar que:

$$L(1-tree) = 11 \leq 13 = w(TSP)$$

por lo que la cota inferior es válida y por lo tanto menor al valor óptimo.

Observación 3.4.1.4. El método del $1-árbol$ es una relajación del TSP. La estructura de un $1-árbol$ es menos restrictiva que la de un circuito hamiltoniano ya que, aunque contenga un número n de aristas como un ciclo hamiltoniano, no se garantiza que forme un ciclo en todos los casos. Por ello, el $1-árbol$ generalmente ofrece un coste menor o igual al valor exacto real de dicho TSP.

Observación 3.4.1.5. La cota inferior es un límite, no una solución exacta. Se utiliza principalmente para descartar soluciones poco prometedoras para el problema. En el caso de que el $1-árbol$ construido diese el mismo resultado que la solución óptima, se podría deducir que dicho $1-árbol$ coincide exactamente con el ciclo hamiltoniano óptimo.

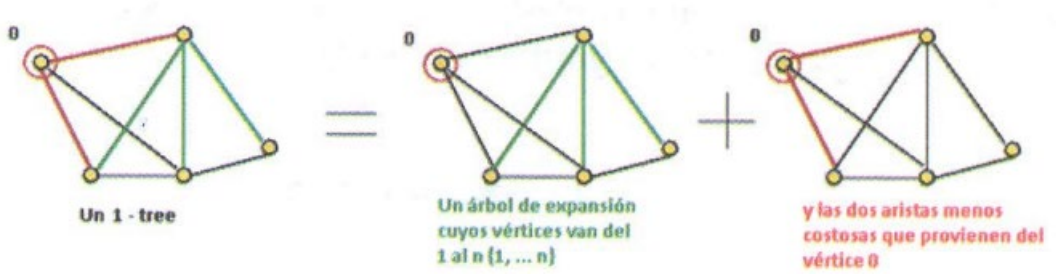

Figura 12. Ejemplo sencillo del método del 1-árbol. [9]

3.4.2 Algoritmo de Christófides.

El algoritmo de Christófides, propuesto en 1976 [1], es el mejor algoritmo aproximado conocido para el TSP en términos de garantía de aproximación. Se trata de un algoritmo $1.5-aproximado$, lo que significa que la solución generada por el algoritmo nunca es más del 50% peor que la óptima.

Observación 3.4.2.1. La existencia de un algoritmo polinomial $\varepsilon - aproximado$ para el TSP, con algún $\varepsilon < 1/2$, sigue siendo un problema abierto.[2]

Dado que el TSP es NP-completo, la resolución exacta del problema en tiempo polinomial es inviable para casos de gran envergadura. En este contexto, Christófides diseñó un método basado en árboles de expansión mínima, emparejamientos de mínimo coste y circuitos eulerianos, proporcionando una solución eficiente y con una cota de error demostrable.

Algoritmo 3.4.2.2.[2] Algoritmo de Christófides. Dado grafo completo $G = (V, E, w)$ con V el conjunto de nodos, E el conjunto de aristas, y $w: E \to \mathbb{R}^+$ una matriz de pesos, el Algoritmo de Christófides genera una solución S para el TSP.

El algoritmo sigue una serie de pasos para la resolución que son los siguientes:

- En primer lugar, se procede a obtener el árbol de mínima expansión sobre el grafo original $G = (V, E, w)$ utilizando el método de Prim, Kruskal o Borůvka dependiendo de las características del grafo. (3.3.1). El árbol obtenido, T, cubre todos los nodos con la menor suma de pesos posible.

$$T = arg_{T' \subseteq E,} \min_{|T'|=|V|-1} \sum_{e \in T'} w(e)$$

Observación 3.4.2.3. Se garantiza que $w(T) \leq w(\pi^*)$ donde $w(\pi^*)$ es el recorrido óptimo del TSP.

- En segundo lugar, se procede a localizar los nodos en T cuyo grado sea impar, donde:

$$V_{impar} = \{v \in V | deg_T(v) es\ impar\}$$

Según el teorema de Euler, siempre hay un número par de vértices de grado impar en cualquier grafo conexo. [15]

- Posteriormente, se procede al emparejamiento de mínimo coste. Se encuentra un emparejamiento perfecto de mínimo coste M para los vértices de grado impar, minimizando:

$$M = arg \min_{M' \subseteq E} \sum_{e \in M'} w(e)$$

este paso se resuelve mediante algoritmos de emparejamiento en grafos bipartitos, con complejidad $O(n^3)$. [2]

En este proceso se garantiza que:

$$w(M) \leq \frac{1}{2} w(\pi^*)$$

- Tras la realización del emparejamiento del paso anterior, se construye el multigrafo G' agregando al MST las aristas de M:

$$G' = (V, T \cup M)$$

Observación 3.4.2.4. Resulta una propiedad clave que todos los vértices tienen grado par, por lo que G' forma un circuito euleriano. [12]

- Finalmente, se obtiene un circuito hamiltoniano eliminando los vértices repetidos del circuito euleriano conseguido en el paso previo, pero conservando el orden del recorrido.
  La desigualdad triangular del garantiza que el peso de la solución final satisface:

$$w(C_h) \leq w(T) + w(M) \leq w(\pi^*) + \frac{1}{2} w(\pi^*)$$

Nota 3.4.2.5. La desigualdad triangular es un principio fundamental en matemática y teoría de grafos que establece que, en un espacio métrico, la distancia directa entre dos puntos nunca es mayor que la suma de las distancias de un camino intermedio. [2]

La complejidad computacional del algoritmo es de $O(n^3)$ lo que resulta mucho más asumible que implementando métodos exactos. Aunque su rendimiento puede ser superado en la práctica por métodos heurísticos, su ventaja principal es la garantía matemática que aporta sobre la calidad de la solución.

Ejemplo 3.4.2.6. Se tiene un repartidor que debe visitar las ciudades A, B, C y D y desea minimizar la distancia recorrida. La matriz de distancias correspondiente a este problema es la siguiente:

	A	B	C	D
A	0	4	8	7
B	4	0	6	3
C	8	6	0	5
D	7	3	5	0

Tabla 4. Matriz de distancias ejemplo 3.3.3.6.

- En primer lugar, se construye el árbol de mínima expansión mediante el algoritmo de Prim, seleccionando las aristas de menor coste sin formar ciclos.
  El MST resultante es el siguiente:
  $$T = \{(B, D), (B, A), (D, C)\}$$
  $$w(T) = 3 + 4 + 5 = 12$$
- Se seleccionan los vértices de grado impar que en este caso son A y C. Estos dos vértices deben de ser emparejados.
- Se busca la arista de menor peso entre los nodos de grado impar A y C. La distancia $A - C$ es 8. Por lo que el emparejamiento mínimo es $w(M) = 8$.
- Se construye el multigrafo euleriano agregando la arista $A - C$ al MST.
  $$G' = \{(B, D), (B, A), (D, C), (A, C)\}$$
  Ahora todos los vértices del grafo tienen grado par, por lo que el grafo es euleriano.
- Un posible recorrido euleriano en el multigrafo es $B \to D \to C \to A \to B$ que al tratarse de un ejemplo tan sencillo coincide que es un circuito hamiltoniano por lo que no será necesario eliminar ningún posible nodo repetido. El resultado será entonces un circuito hamiltoniano con peso:
  $$w(C_h) = 3 + 5 + 8 + 4 = 20$$

En este caso al tratarse de un problema tan sencillo la solución obtenida mediante el algoritmo de Christófides coincide con la que se obtiene mediante el método de fuerza bruta. Por este motivo podemos ver que se cumple la regla de que al implementar dicho algoritmo se garantiza la obtención de una solución con una cota de error ε como máximo del 50%.

La complejidad computacional de este método es de $O(n^3)$ debido a que su paso que genera un mayor esfuerzo es en el que se lleva a cabo el emparejamiento de mínimo coste. Resulta más eficiente que los métodos exactos cuya complejidad es de $O(n!)$. La implementación de este método es viable para problemas de tamaño moderado. Para problemas de una mayor envergadura, empieza a ser ineficiente y se requiere la implementación de métodos heurísticos, computacionalmente más sencillos.

3.5 Métodos heurísticos

Los métodos heurísticos son estrategias de resolución de problemas que permiten encontrar soluciones aproximadas en tiempos computacionales reducidos. Se utilizan especialmente en problemas de optimización combinatoria donde los algoritmos exactos resultan impracticables debido a la complejidad computacional. Estos métodos no garantizan la obtención de la solución óptima, pero buscan soluciones razonablemente buenas en un tiempo aceptable. Su eficacia radica en la aplicación de reglas intuitivas, exploración de soluciones parciales y técnicas iterativas de mejora. A diferencia de los métodos aproximados, no aportan un coeficiente de error ε determinado al que atenerse por lo que resultan de menor fiabilidad. [16]

3.5.1 Cota superior. Método de inserción más cercana.

El método de inserción más cercana es una heurística de construcción para aproximar soluciones del TSP. La estrategia consiste en construir iterativamente un ciclo hamiltoniano, agregando en cada paso el vértice no visitado más cercano a la solución parcial actual. Este método pertenece a la familia

de los algoritmos de inserción, en los cuales se parte de un ciclo inicial y se extiende gradualmente mediante la inserción de vértices. Es una técnica intuitiva y eficiente en términos de complejidad computacional, aunque no garantiza una solución óptima.

Algoritmo 3.5.1.1. Método de inserción más cercana [2]. Dado un grafo completo $G = (V, E, w)$, donde $V = \{v_1, v_2, \dots, v_n\}$ es el conjunto de vértices del grafo, E es el conjunto de aristas con pesos asociados w dónde $E \to \mathbb{R}^+$, cumpliendo la desigualdad triangular (Nota 3.4.2.5.) $w(u, v) \leq w(u, x) + w(x, v)$ para todos los vértices $u, v, x \in V$.

Este método consta de los siguientes pasos:

- En primer lugar; durante la fase de inicialización, se escoge de manera arbitraria un vértice de inicio $v_1 \in V$ y posteriormente se selecciona un segundo vértice v_2 que ha de ser el vértice perteneciente a V más cercano al vértice v_1, es decir:
$$v_2 = arg \min_{v \in V \setminus \{v_1\}} w(v_1, v)$$
De esta forma, el ciclo inicial que se genera será $C_2 = (v_1,\ v_2, v_1)$.

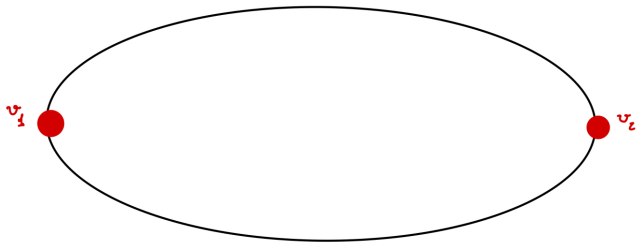

Figura 13. Elaboración propia curso 2025. Ciclo inicial $\boldsymbol{C_2}$.

- Se procede a la selección del siguiente vértice. Para cada iteración k, se selecciona el vértice v_k más cercano a algún vértice en el ciclo actual C_{k-1}, es decir:
$$v_k = arg \min_{v \in V \setminus C_{k-1}} \min_{u \in C_{k-1}} w(u, v)$$
donde C_{k-1} representa el conjunto de vértices ya insertados en el ciclo.
- Se inserta el nodo en la posición óptima. Se busca la posición óptima en el ciclo para insertar el vértice v_k. Esto se hace determinando el par de vértices (v_i, v_j) consecutivos en C_{k-1} que minimicen el incremento de coste al insertar v_k:
$$(v_i, v_j) = arg \min_{(v_i, v_j) \in C_{k-1}} (w(v_i, v_k) + w(v_k, v_j) - w(v_i, v_j))$$
Al realizar este paso, el ciclo se actualiza:
$$C_k = C_{k-1} \cup \{(v_i, v_k), (v_k, v_j)\} \setminus \{(v_i, v_j)\}$$
- Se procede a repetir el proceso hasta que la totalidad de los vértices de V han sido insertados en el ciclo, obteniendo así el ciclo hamiltoniano deseado.

Ejemplo 3.5.1.2. Se considera un conjunto de ciudades A, B, C, D cuya matriz de distancias es la siguiente:

$$\begin{bmatrix} 0 & 4 & 8 & 7 \\ 4 & 0 & 6 & 3 \\ 8 & 6 & 0 & 5 \\ 7 & 3 & 5 & 0 \end{bmatrix}$$

- En primer lugar, se selecciona el nodo A de forma arbitraria para inicializar el ciclo. Posteriormente se toma la arista incidente en A de menor coste que en este caso es la arista que

le une al nodo B. El valor de esta arista es $w(A,B) = 4$. El ciclo inicial resultante será el siguiente:

$$C_2 = (A, B, A)$$

- Se selecciona el vértice más cercano a A o a B que en este caso es D desde B, cuyo coste es $w(B,D) = 3$.
- Se busca cual es la mejor opción para proceder a la inserción de D entre cada par de vértices de C_2. Al ser un problema de TSP simétrico, será indiferente donde introducir el nodo D ya que la arista entre A y B será igual a la que hay entre B y A. Al insertar D se obtiene:
  $\Delta w = w(A,D) + w(D,B) - w(A,B) = w(B,D) + w(D,A) - w(B,A) = 7 + 3 - 4 = 6$
  El ciclo C_3 resultante que se escoge es el siguiente:
  $$C_3 = (A, D, B, A)$$
- Se lleva a cabo en este caso la última inserción que en este caso es la del nodo C. Se ha de probar a insertarlo entre A y D, entre D y B y entre B y A.
  En este caso la posición óptima es la inserción de C entre A y D ya que:
  $$\Delta w = w(A,C) + w(C,D) - w(A,D) = 8 + 5 - 7 = 6$$
  Esta posición es la que consigue la posición más óptima entre las 3 posibles.
- El ciclo hamiltoniano resultante C_4, es ya la solución final que implica este circuito:
  $$A \to C \to D \to B \to A$$
  Su coste total es:
  $$w(A,C) + w(C,D) + w(B,A) = 8 + 5 + 3 + 4 = 20$$

En este caso, esta solución es la misma que la resultante empleando métodos exactos. Esto ocurre por la sencillez del ejemplo, pero no tiene porqué darse este caso.

La complejidad computacional de este método es de $O(n^2)$ lo que lo hace más rápido que los métodos exactos cuya complejidad es de $O(n!)$ y que métodos aproximados como el método de Christófides; previamente mencionado en este trabajo, cuya complejidad es de $O(n^3)$. El problema de la implementación de métodos heurísticos como este es la falta de garantías en cuanto a la solución aportada. Al contrario que los métodos aproximados, no arrojan ninguna tasa de error ε que muestre de alguna manera la fiabilidad de la solución aportada. Este método es excesivamente sensible a la elección del vértice inicial, ya que hace enormemente diferir en muchos casos las diversas soluciones.

3.5.2 Método del árbol.

El método del árbol es una heurística basada en árboles de expansión mínima, utilizada para encontrar soluciones aproximadas al Problema del Viajante de Comercio. Su enfoque se basa en aprovechar la estructura de los árboles de expansión mínima para generar un recorrido hamiltoniano que conecta todos los nodos con un coste relativamente bajo.

Este método se fundamenta en la observación de que un árbol de expansión mínima es una estructura eficiente en términos de costes de conexión, aunque no necesariamente genere un circuito hamiltoniano. Para convertir esta estructura en una solución factible para el TSP, el método del árbol introduce un proceso de transformación basado en la duplicación de aristas y la generación de un recorrido euleriano, que luego se convierte en un ciclo hamiltoniano eliminando visitas repetidas a los nodos.

Dado que la obtención de un MST puede realizarse en tiempo polinomial mediante algoritmos previamente definidos, como Prim y Kruskal, el método del árbol ofrece una solución computacionalmente eficiente para el TSP. Sin embargo, a diferencia de otros algoritmos de aproximación como el de Christófides, este método no garantiza una solución con un factor de aproximación ε mejor que 2 respecto al óptimo. A pesar de sus limitaciones, el método del árbol es una herramienta útil en escenarios donde la rapidez es prioritaria sobre la precisión. Su simplicidad y

eficiencia lo convierten en una alternativa práctica para resolver variantes grandes del problema en tiempos razonables.

Algoritmo 3.5.2.1. Algoritmo del árbol. Dado un grafo completo $G = (V, E, w)$. Siendo V, E y w lo mismo que en el Algoritmo 3.5.1.1. [2]

Para la puesta en práctica de este método se siguen los siguientes pasos:

- En primer lugar, se procede a la construcción del árbol de expansión mínima T respectivo al grafo completo G, utilizando uno de los métodos expuestos en 3.3.

$$T = arg \min_{T' \subseteq E, |T'| = |V|-1} \sum_{e \in T'} w(e)$$

donde $w(e)$representa el coste de la arista e y T es un conjunto de $|V| - 1$ aristas que conecta todos los vértices sin ciclos.

- A continuación, se duplican las aristas del MST. Se crea un multigrafo G' a partir de T. Esto se consigue duplicando la totalidad de sus aristas. Esto asegura que el grafo resultante sea euleriano; lo que implica que todos los vértices tienen grado par.

$$E(G') = \{e | e \in T, cada\ e\ aparece\ dos\ veces\}.$$

Nota 3.5.2.2. Este paso es clave, pues permite que exista un circuito que pase por todas las aristas exactamente una vez, siguiendo el teorema de Euler. [17]

- Se determina un recorrido euleriano en G'; es decir, un circuito que visita cada arista exactamente una vez. Este recorrido se puede obtener en tiempo $O(n)$ con un procedimiento adecuado como el algoritmo de Hierholzer. [18]
- Se convierte el recorrido euleriano en un ciclo hamiltoniano, eliminando así los vértices repetidos. Si el recorrido euleriano es:

$$C = (v_1, v_2, \dots, v_k, v_1)$$

se transforma en un tour hamiltoniano π conservando solo la primera aparición de cada vértice:

$$\pi = (v_1, v_{i_2}, \dots, v_{i_n}, v_1)$$

donde i_1, i_2,…, i_n son índices que garantizan que cada vértice aparece una única vez.

Nota 3.5.2.3. Gracias a la desigualdad triangular, la distancia total del ciclo no supera el doble de la distancia óptima (π^*):

$$w(\pi) \leq 2w(\pi^*)$$

Ejemplo 3.5.2.4. Dada la siguiente matriz de pesos:

$$\begin{bmatrix} 0 & 2 & 9 & 10 \\ 2 & 0 & 6 & 4 \\ 9 & 6 & 0 & 3 \\ 10 & 4 & 3 & 0 \end{bmatrix}$$

- En primer lugar, se construye el MST, obteniendo que:

$$T = \{(A, B), (B, D), (C, D)\}$$

con un coste total:

$$w(T) = 2 + 4 + 3 = 9$$

- Se duplican las aristas hasta que el grafo G' contiene:

$$(A, B), (A, B), (B, D), (B, D), (C, D), (C, D)$$

- Se selecciona un recorrido euleriano utilizable que en este caso será:

$$C = A \to B \to D \to C \to D \to B \to A$$

- Eliminando las repeticiones, el ciclo hamiltoniano resultante será el siguiente:

$$\pi = A \to B \to D \to C \to A$$

cuyo peso total será:

$$w(\pi) = 2 + 4 + 3 + 9 = 18$$

Algoritmo 3.5.2.5. Algoritmo de Hierholzer. Un grafo tiene un ciclo euleriano si y sólo si es conexo y cada vértice tiene grado par. [18]

El algoritmo de Hierholzer es un procedimiento eficiente para encontrar un circuito euleriano en un grafo euleriano; es decir, un recorrido que visita cada arista exactamente una vez y regresa al punto de inicio. Este algoritmo es de gran importancia en la implementación del método del árbol y del método de Christófides para el TSP, ya que permite transformar un árbol de expansión mínima duplicado en un recorrido euleriano, que posteriormente se convierte en un ciclo hamiltoniano eliminando nodos repetidos.

Dado un grafo $G = (V, E)$ los pasos que se siguen en la implementación de este algoritmo son los siguientes:

- Se escoge un vértice de inicio arbitrario $v_0 \in V$.
- Se construye un subcircuito parcial que recorra el grafo partiendo de v_0, eligiendo en cada paso una arista no visitada. Al recorrer una arista e, esta se elimina temporalmente del grafo. Se continúa recorriendo el grafo hasta regresar a v_0, formando un circuito parcial:

$$C = (v_0, v_1, \dots, v_k, v_0)$$

- Se expande el circuito parcial. Si todavía quedan aristas por visitar, se escoge un vértice v_i en C que disponga de aristas disponibles. Desde v_i, se construye un nuevo subcircuito hasta que regrese a v_i. Este subcircuito ha de insertarse en el recorrido existente.
- Se repiten los pasos 2 y 3 hasta que todas las aristas hayan sido visitadas y el circuito sea completo. El resultado, es un circuito euleriano que visita cada arista exactamente una vez.

4 RESULTADOS / VALIDACIÓN / PRUEBA

En este apartado del trabajo, se pretende comparar los métodos previamente expuestos en este trabajo aplicados a un caso real, donde la premisa principal es que a través del planteamiento del problema como un problema TSP, se logren sacar conclusiones relevantes sobre los diferentes algoritmos y sus ventajas y desventajas en cuanto a tiempo de computación y nivel de aproximación. Con este caso práctico, se tratará de optimizar el recorrido de una patrulla aérea que conecte núcleos logísticos importantes en la península con las áreas de soberanía nacional en el norte de África. La planificación eficiente de rutas es un problema relevante en múltiples ámbitos, desde la logística hasta la defensa y la seguridad. En este estudio, la patrulla en la que nos centramos pretende optimizar la ruta de una patrulla aérea que parte de la base aérea del Ejército del Aire y del Espacio de Morón y debe visitar una serie de localizaciones estratégicas antes de regresar a su punto de origen.

La patrulla aérea debe recorrer 10 puntos estratégicos, incluyendo bases militares y posiciones geográficas clave en el ámbito de la defensa nacional. La patrulla tras salir de la base de Morón a de visitar la Base Naval de Rota, el Arsenal de Cartagena, las dos ciudades autónomas, Ceuta y Melilla, la Isla de Alborán, el Peñón de Vélez de la Gomera, las Islas Chafarinas, y las ciudades del sur peninsular de Málaga y Almería. La principal razón de ser de este tipo de patrulla, es hacer presencia en territorios pertenecientes a la soberanía nacional y detectar operaciones de narcotráfico o tráfico de migrantes en las costas de Cádiz, estrecho de Gibraltar y mediterráneo occidental y mar de Alborán. [19]

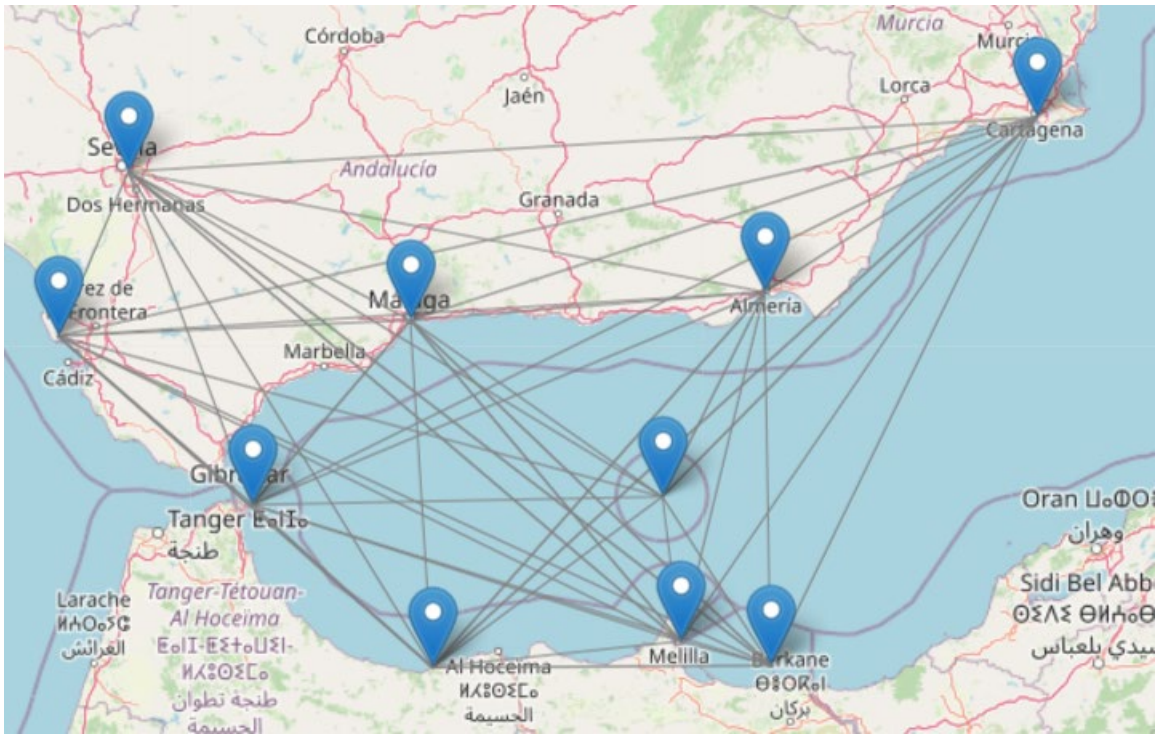

Figura 14. Elaboración propia curso 2025, herramienta OpenStreetMap. Grafo superpuesto al mapa de la zona.

Cada una de estas localizaciones se han definido por sus coordenadas geográficas sacadas con la herramienta para la navegación Open CPN. Se exponen en la siguiente tabla:

	LATITUD	LONGITUD
MORÓN	37,3597	-5,9675
ROTA	36,6394	-6,3460
CEUTA	35,8990	-5,2805
MÁLAGA	36,7131	-4,4128
PEÑÓN DE VÉLEZ	35,1778	-4,2997
ISLA DE ALBORÁN	35,9377	-3,0365
MELILLA	35,2944	-2,9336
CHAFARINAS	35,1809	-2,4339
CARTAGENA	37,5954	-0,9812
ALMERÍA	36,8343	-2,4678

Tabla 5. Elaboración propia curso 2025 con datos sacados de la herramienta Open CPN. Los valores se muestran en grados siendo positivos los valores de latitud norte y los valores de longitud este.

Con esta serie de puntos definidos se procede a generar un grafo que muestre de una forma visual la disposición de estos y la unión de todos ellos con cada uno de los demás, generando así las aristas que posteriormente formarán parte de los distintos posibles circuitos seleccionables para la búsqueda de soluciones de este problema. Al tratarse de un problema que implica 10 ciudades utilizando la expresión

$$\frac{(n-1)!}{2}$$

podemos concluir que el número de ciclos hamiltonianos que sirvan como solución para este circuito es de 181440. Entre todos ellos, la situación **idónea** es encontrar el ciclo de menor peso, es decir:

$$min \sum_{(i,j)\in E} w_{ij}x_{ij}$$

siendo E el conjunto de aristas, y sabiendo que a cada arista (i,j) se le asocia un peso w_{ij} que en este caso representa la distancia entre los puntos i y j.

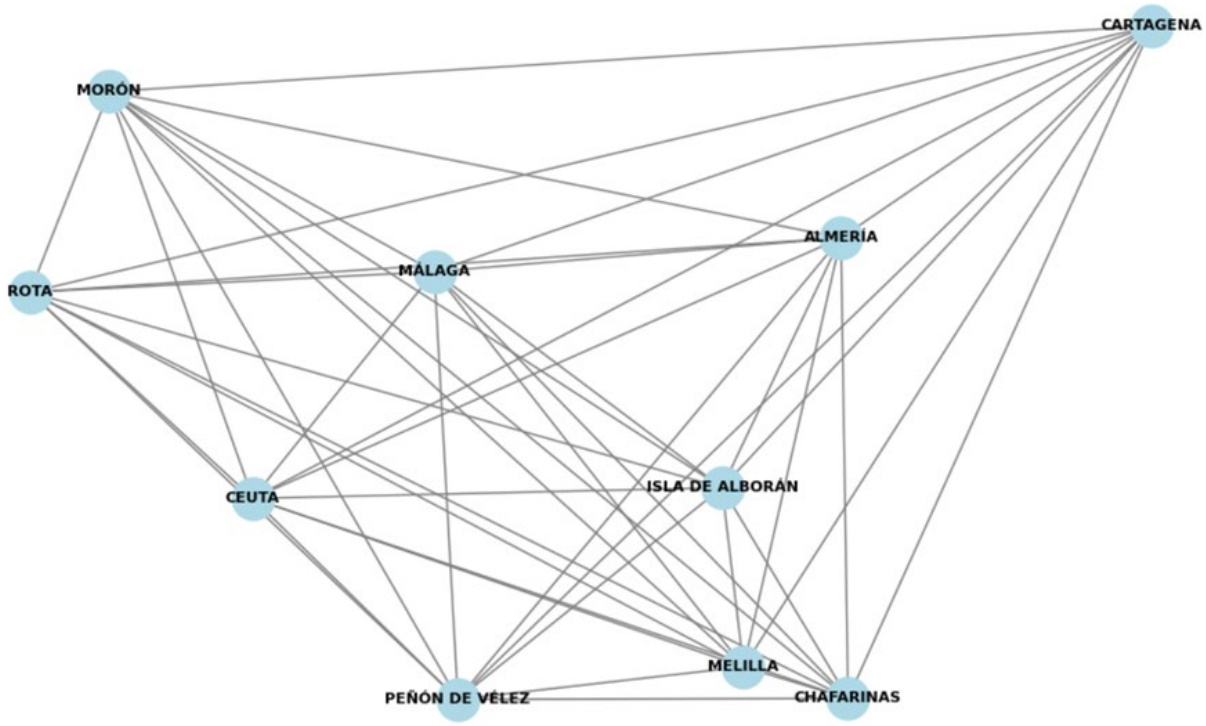

Figura 15. Elaboración propia curso 2025 con la herramienta SageMath. Grafo que expone la situación de los nodos y la posible conectividad de cada uno de ellos con el resto de nodos.

Para la resolución del problema, utilizamos una matriz de distancias que contiene las distancias geodésicas entre cada par de ubicaciones. Esta matriz nos permitirá evaluar todas las posibles rutas y determinar la más eficiente. Los datos de la latitud y longitud de cada ciudad se utilizarán para calcular estas distancias, utilizando la fórmula de Haversine. [20] Los datos de la matriz, con la implementación de esta fórmula, se han sacado también mediante la utilización del programa SageMath.

UNIDADES (MN	MORÓN	ROTA	CEUTA	MÁLAGA	PEÑÓN DE	ISLA DE AL	MELILLA	CHAFARINAS	CARTAGEN	ALMERÍA
MORÓN	0	46,84	93,59	84,14	153,73	165,17	192,2	215,46	238,5	170,9
ROTA	46,84	0	68,12	93,41	132,72	165,94	184,69	209,72	263,7	187,39
CEUTA	93,59	68,12	0	64,42	64,6	109,38	120,41	145,87	230,86	147,36
MÁLAGA	84,14	93,41	64,42	0	92,15	81,3	111,4	133,11	172,87	94,03
PEÑÓN DE VÉLEZ	153,73	132,72	64,6	92,15	0	76,78	67,5	91,76	216,37	133,42
ISLA DE ALBORÁN	165,17	165,94	109,38	81,3	76,78	0	38,87	54,1	140,29	60,37
MELILLA	192,2	184,69	120,41	111,4	67,5	38,87	0	25,48	167,14	95
CHAFARINAS	215,46	209,72	145,87	133,11	91,76	54,1	25,48	0	160,87	99,07
CARTAGENA	238,5	263,7	230,86	172,87	216,37	140,29	167,14	160,87	0	84,59
ALMERÍA	170,9	187,39	147,36	94,03	133,42	60,37	95	99,07	84,59	0

Tabla 6. Elaboración propia curso 2025. Las distancias se expresan en millas náuticas. Los datos se han tomado mediante SageMath a través de las coordenadas.

Nota 4.0.0.1. Las distancias se expresan en Millas Náuticas en todo el apartado.

4.1 Resolución por métodos exactos

Al tratarse de un problema de 10 ciudades, es computacionalmente viable resolverlo por fuerza bruta. Utilizando la herramienta SageMath (código en anexo II) se ha hallado que la solución óptima al problema es de $\mathbf{728,61}\ \boldsymbol{MN}$ siguiendo el siguiente ciclo:

$$Morón \rightarrow Rota \rightarrow Ceuta \rightarrow Peñón\ de\ Vélez \rightarrow Melilla \rightarrow Chafarinas \rightarrow Isla\ de\ Alborán \rightarrow Almería \rightarrow Cartagena \rightarrow Málaga \rightarrow Morón$$

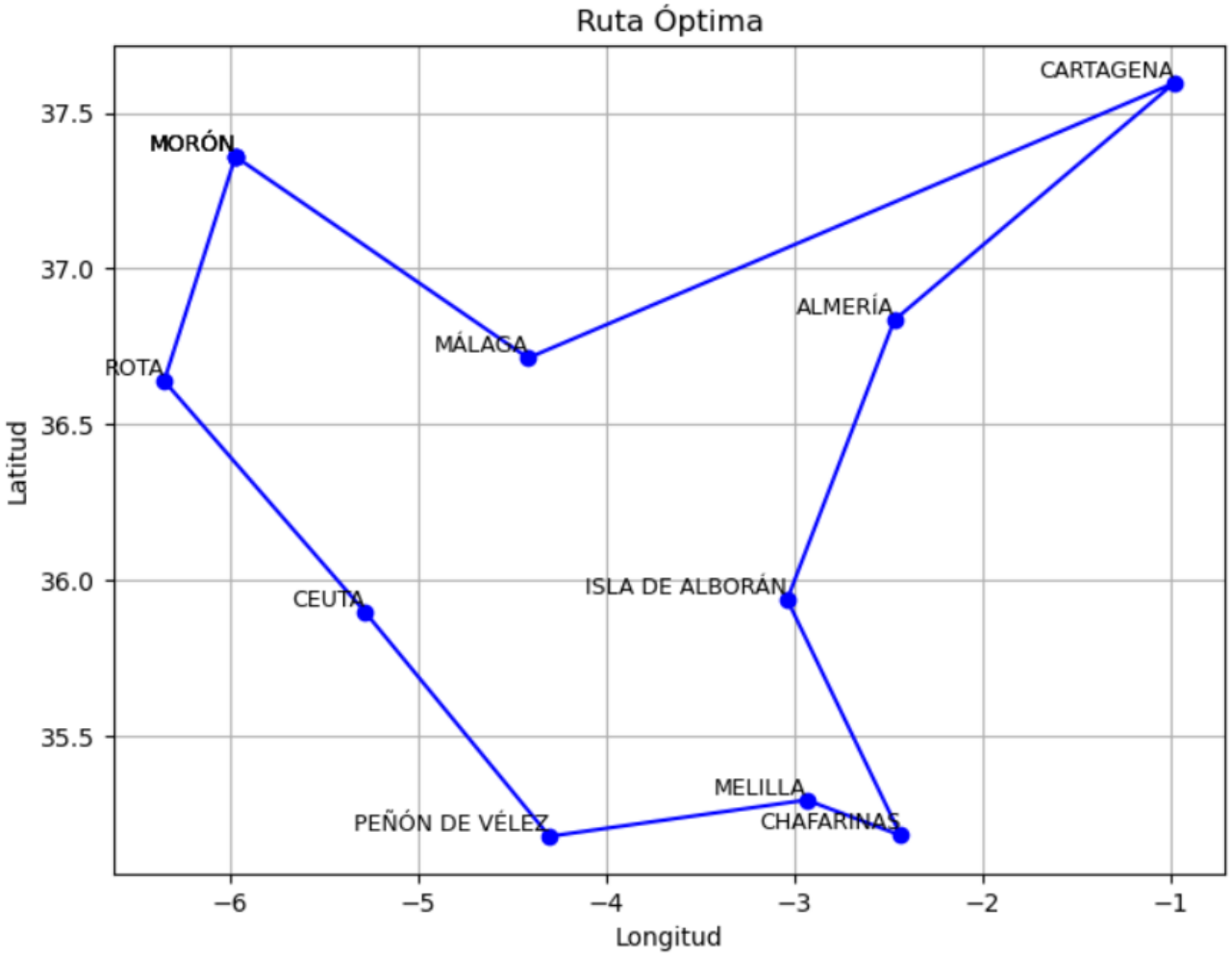

Figura 16. Elaboración propia curso 2025. Generado con SageMath. Este grafo muestra el circuito hamiltoniano correspondiente a la solución exacta de nuestro TSP.

Se ha empleado este resultado para comparar la eficiencia del resto de métodos y su cercanía o lejanía a este resultado. Cabe destacar que en entornos de alto nivel como SageMath, el método de fuerza bruta suele volverse impracticable alrededor de 11 o 12 ciudades debido a la explosión combinatoria.[21] Es decir, para 10 ciudades podría ser manejable, pero a partir de 11 o 12, ya el cálculo exacto mediante fuerza bruta es generalmente inviable. Esto resulta refutado por propia experiencia durante la realización del trabajo. Se ha escogido un problema con 10 ciudades con la intención de poder hallar este resultado exacto para poder hacer comparaciones con la eficiencia de los diferentes métodos.

4.2 Resolución por métodos aproximados

4.2.1 Cálculo de una cota inferior. Método del $1-\acute{a}rbol$.

Para la resolución se toma que el grafo completo del cual se dispone es $G=(V,E)$, donde V es el conjunto de ciudades y E, es el conjunto de aristas.

- El primer paso será eliminar Morón del grafo, por lo que el grafo que se queda es:
$$G'=(V\backslash\{Morón\},E')$$
- Se construye un árbol de mínima expansión en G', mediante el algoritmo de Prim obteniendo el siguiente resultado:

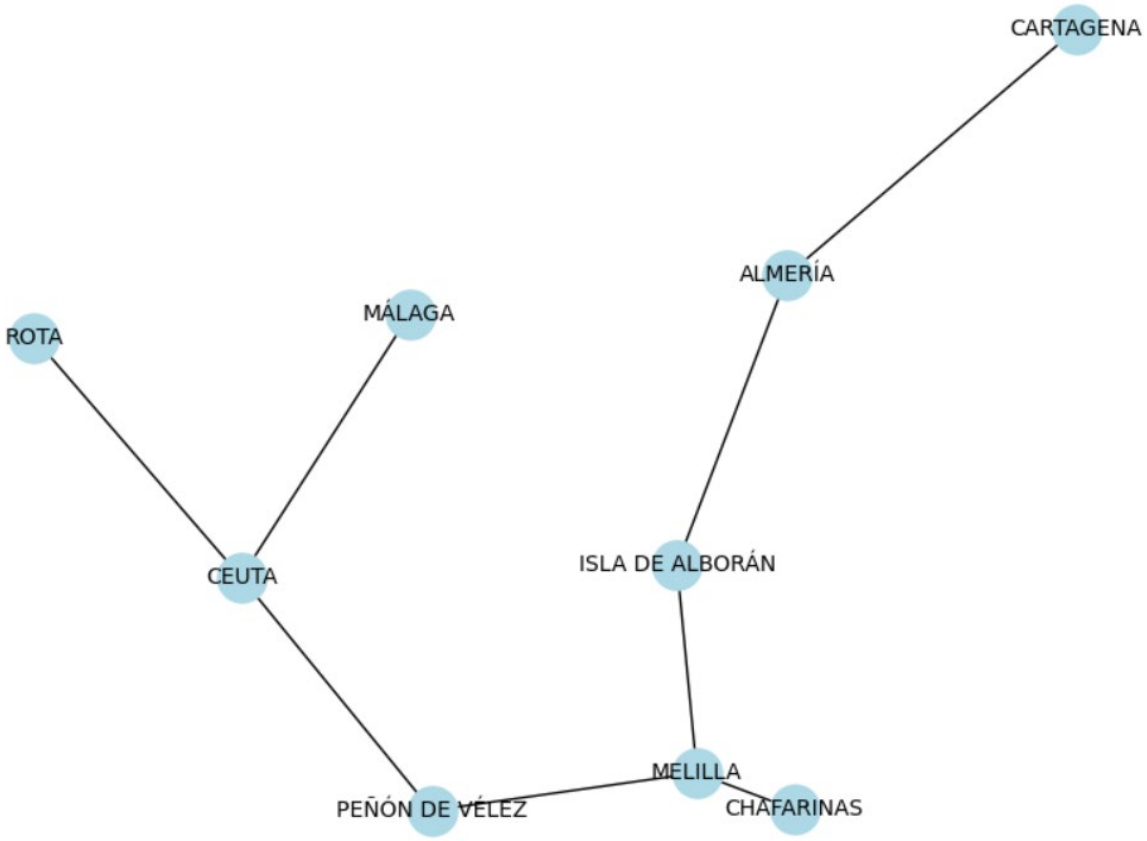

Figura 17. Elaboración propia curso 2025 a través de la herramienta SageMath. Árbol de mínima expansión utilizado para obtener una cota inferior con el método del $\mathbf{1-\acute{a}rbol}$

El coste total de este árbol de mínima expansión es de $473{,}97\ MN$.

- Se seleccionan las dos aristas de menor coste que unen Morón con algún vértice de este árbol de mínima expansión, en este caso son:
$Morón \rightarrow Rota$ con un coste de $46{,}84MN$.
$Morón \rightarrow Málaga$ con un coste de $84{,}14MN$.
- Se suman el coste de las dos aristas del paso anterior con el coste del MST sin Morón previamente calculado.
$$w(total)=w(MST-Morón)+w(Morón\rightarrow Rota)+w(Morón\rightarrow Málaga)$$
$$w(total)=473{,}97+46{,}84+84{,}14=\mathbf{604,95}\boldsymbol{MN}$$
siendo w referente a los costes.

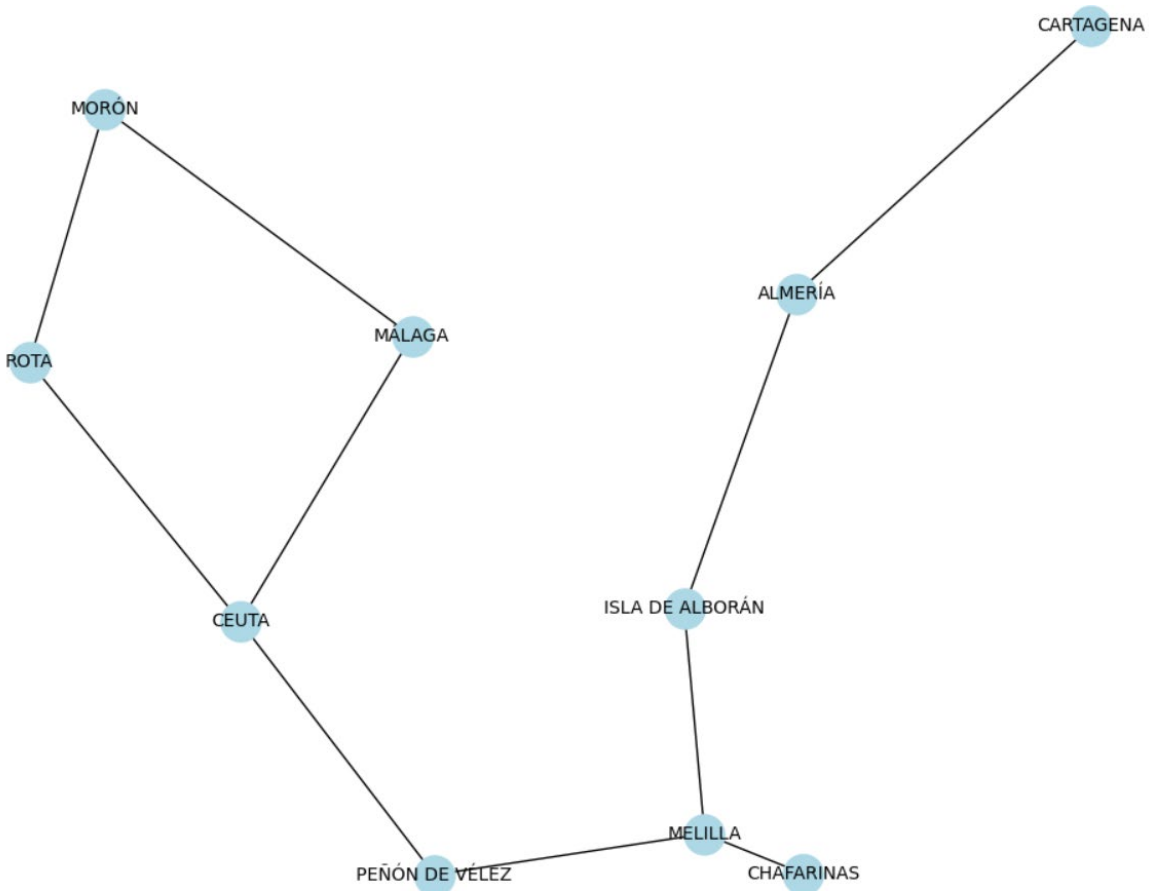

Figura 18. Elaboración propia curso 2025 a través de la herramienta SageMath. Ilustración del resultado del método del $\mathbf{1-}$ á$\boldsymbol{rbol}$

Se puede ver que este resultado es menor a la solución óptima. Cabe destacar que este método no aporta una posible solución al problema, ya que el resultado no es un circuito hamiltoniano, simplemente da una cota inferior que ayuda a delimitar la solución que se puede obtener con otros métodos.

Este método puede dar diferentes resultados dependiendo el nodo que se decida aislar en el primer paso. Por ejemplo, si se decide aislar el Arsenal de Cartagena en vez de la base aérea de Morón, las dos aristas incidentes en ella de mínimo peso, serán la que le une a Almería $(84{,}4MN)$ y la que le une a la Isla de Alborán $(140{,}2MN)$. Si a estas dos aristas se le suma el MST resultante creado con el resto de nodos de valor $435{,}8MN$; la cota inferior resultante será de $\boldsymbol{660{,}4MN}$

Este resultado es de mayor coste que el obtenido eliminando Morón del MST lo que hace que sea más cercano al resultado óptimo final y por tanto sea una cota inferior más ajustada.

4.2.2 Algoritmo de Christófides.

En este caso, se puede aplicar este algoritmo para lograr encontrar una ruta eficiente que minimice la distancia total de la patrulla aérea partiendo desde Morón, como se ha establecido previamente. Este algoritmo sabemos que nos aportará un circuito hamiltoniano que sí será solución a este problema y se sabe que como máximo nos dará un resultado un 50% mayor al resultado óptimo calculado. Podemos concluir que la distancia más elevada que debería dar este método aplicado a este problema será:

$$728{,}61MN \times 1{,}5 = 1092{,}915MN$$

siendo así el error máximo acumulable $\varepsilon = 364{,}305MN$, por tanto:

$$\varepsilon \leq 0{,}5 \times w(solución\ óptima)$$

Este es el método que asegura una solución supeditada a un error menor de todos los métodos de aproximación conocidos.

1) El primer paso para esta resolución será mediante el método de Prim, y apoyado por el programa SageMath, crear el árbol de mínima expansión:

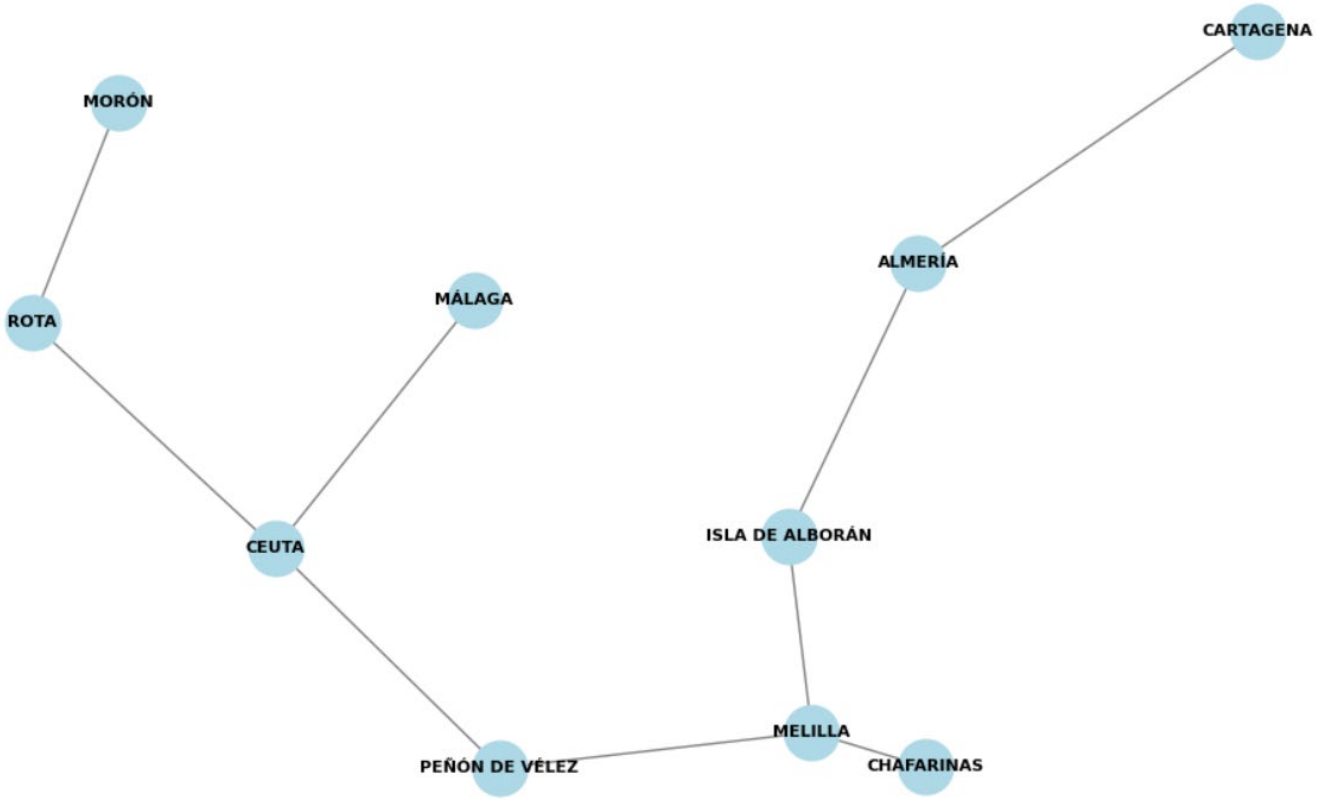

Figura 19. Elaboración propia curso 2025 con el programa SageMath. Árbol de mínima expansión del ejercicio propuesto.

El coste total del MST es de $520{,}8MN$, lo cual no es especialmente relevante para la aplicación de este método, pero que posteriormente puede resultar de utilidad.

2) En segundo lugar, se procede a identificar los nodos de grado impar existentes en el MST, que en este caso son:

$$Morón$$
$$Ceuta$$
$$Málaga$$
$$Melilla$$
$$Chafarinas$$
$$Cartagena$$

3) Teniendo en cuenta estos nodos de grado impar, se realizan los emparejamientos perfectos de mínimo coste M mediante algoritmos de emparejamiento. Estos son los emparejamientos hallados:

$$Chafarinas \rightarrow Melilla$$
$$Málaga \rightarrow Cartagena$$
$$Ceuta \rightarrow Morón$$

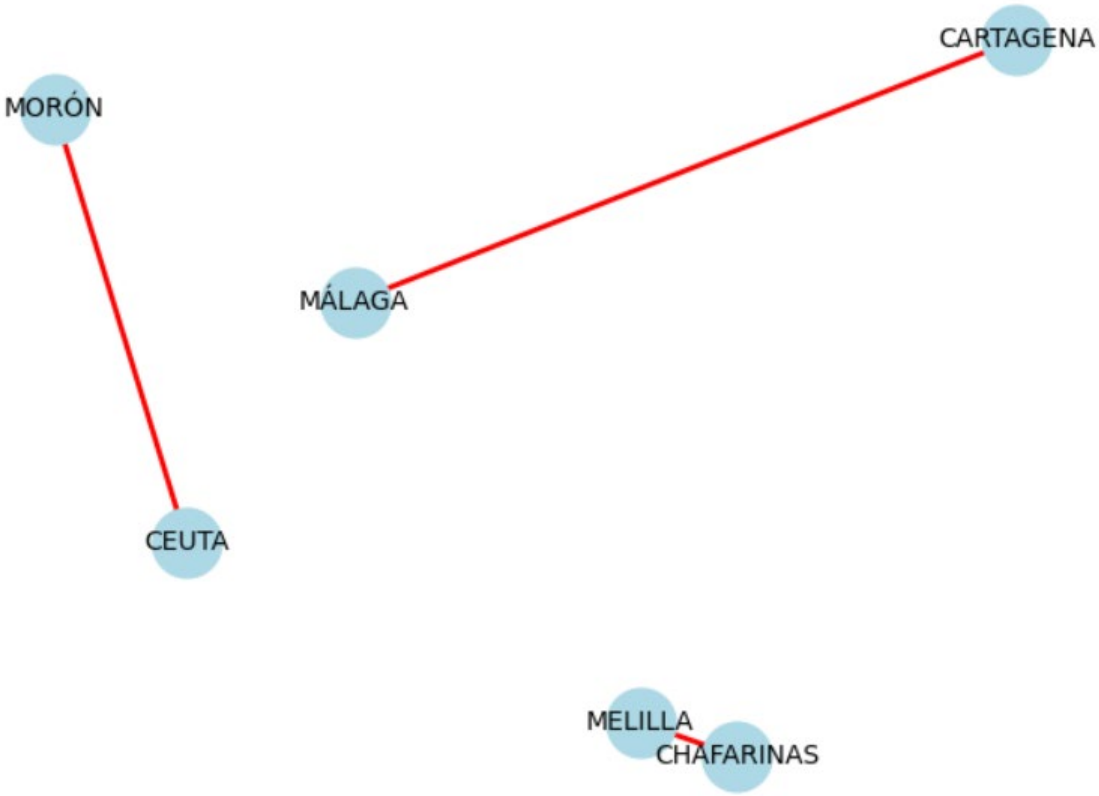

Figura 20. Elaboración propia curso 2025 con el programa SageMath. Emparejamiento perfecto mínimo en nodos impares.

4) Disponiendo del MST y de los emparejamientos perfectos mínimos de los nodos impares se genera un multigrafo G' que incluya ambas. Por lo tanto, se tendrá que:

$$G' = (V, T \cup M)$$

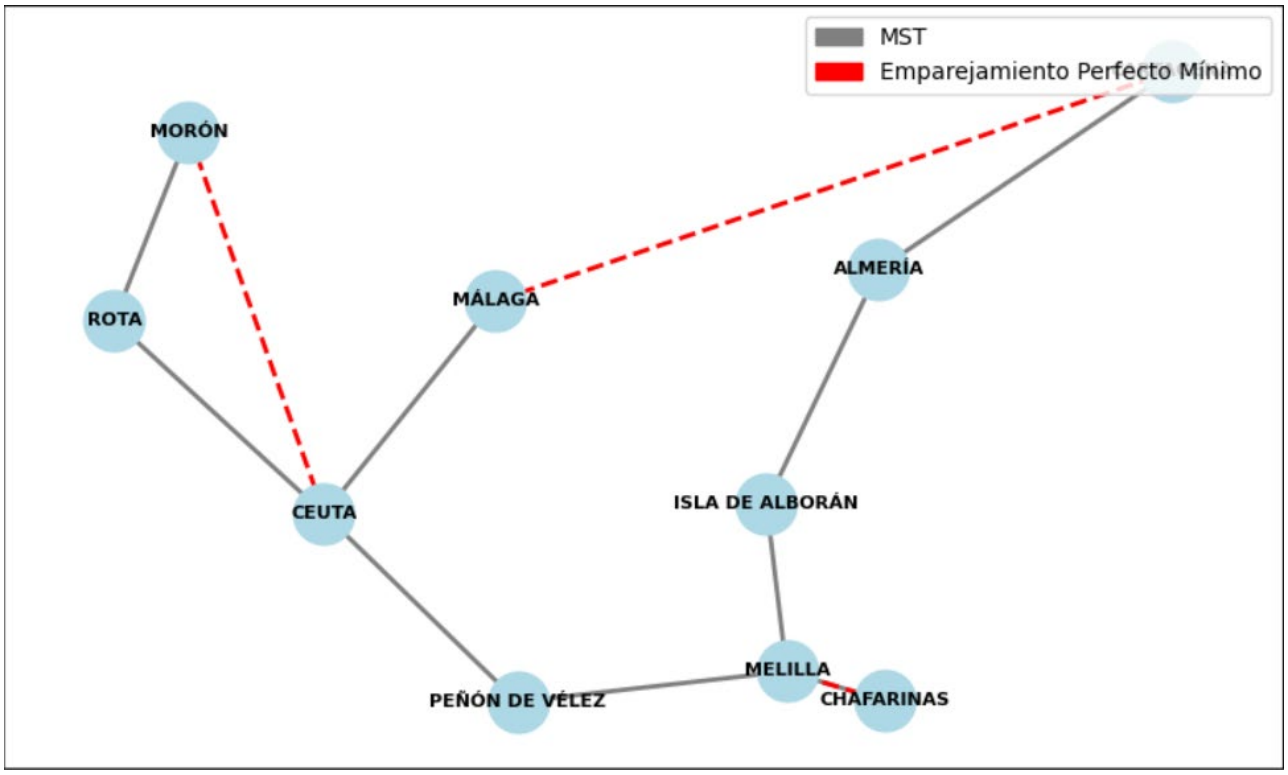

Figura 21. Elaboración propia curso 2025 con programa SageMath. Multigrafo: MST + Emparejamiento Perfecto Mínimo.

Al construir este multigrafo, se obtiene una estructura en la que todos los nodos tienen grado par, lo que garantiza el cumplimiento de una de las condiciones fundamentales establecidas en el teorema de Euler. Este teorema establece que un grafo conexo posee un circuito euleriano con la condición de que todos sus vértices tienen grado par, lo que nos permite asegurar la existencia de un recorrido que atraviesa todas las aristas exactamente una vez.

En este contexto, la estructura resultante permite la construcción de un circuito euleriano que abarca todos los puntos estratégicos de la patrulla aérea, garantizando así que cada ubicación es visitada al menos una vez dentro del recorrido. Sin embargo, dado que el circuito euleriano está basado en la duplicación de ciertas aristas para mantener la paridad de los grados de los vértices, en algunos casos puede ser necesario recorrer ciertas conexiones en más de una ocasión antes de poder derivar un ciclo hamiltoniano.

El grafo euleriano resultante, será el siguiente:

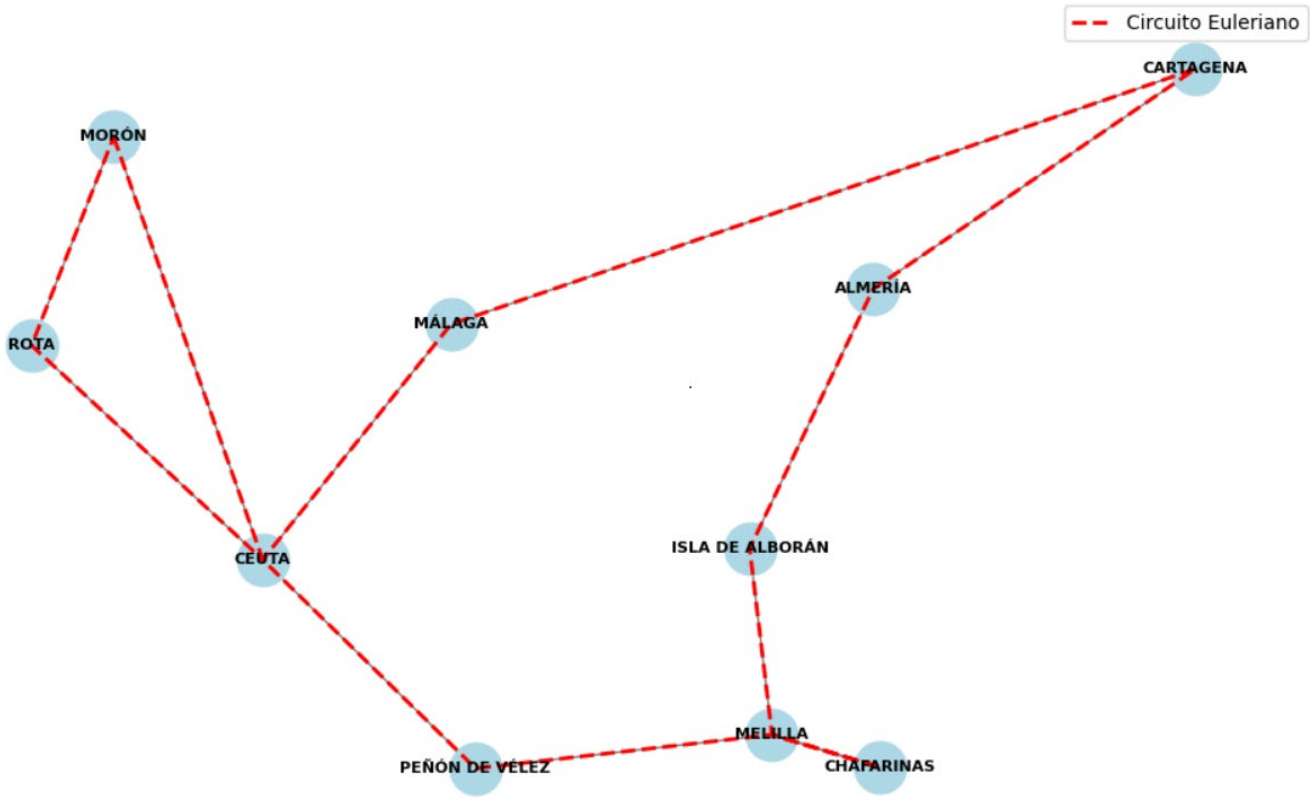

Figura 22. Elaboración propia curso 2025 con programa SageMath. Circuito euleriano.

5) A continuación, para transformar el circuito euleriano en un circuito hamiltoniano, se aplica una estrategia basada en la eliminación de nodos repetidos mientras se preserva la estructura del recorrido. El circuito euleriano, obtenido a partir del multigrafo generado en el algoritmo de Christófides, garantiza que todas las aristas sean recorridas al menos una vez, pero debido a la duplicación de ciertas conexiones para asegurar la paridad de los grados de los vértices, es posible que algunas ubicaciones sean visitadas múltiples veces, como en este caso, por ejemplo, Melilla. Para convertirlo en un ciclo hamiltoniano válido, se recorre la secuencia del circuito euleriano y se agregan los nodos a la ruta hamiltoniana únicamente la primera vez que aparecen, omitiendo las repeticiones posteriores. Este proceso se realiza utilizando una estructura de datos eficiente como un conjunto para rastrear los nodos ya visitados y una lista ordenada para construir la secuencia final. Una vez que todos los nodos han sido visitados sin repeticiones, se cierra el ciclo conectando el último nodo con el nodo de inicio, asegurando así un recorrido hamiltoniano válido y óptimo que respeta la estructura del problema del viajante de comercio. Además, gracias a la desigualdad triangular, la distancia total del circuito hamiltoniano final es, como máximo, 1,5 veces la solución óptima. [22]

   El ciclo hamiltoniano resultante es el siguiente:

$$Morón \rightarrow Ceuta \rightarrow Peñón\ de\ Vélez \rightarrow Melilla \rightarrow Chafarinas \rightarrow Isla\ de\ Alborán \rightarrow Almería \rightarrow Cartagena \rightarrow Málaga \rightarrow Rota \rightarrow Morón$$

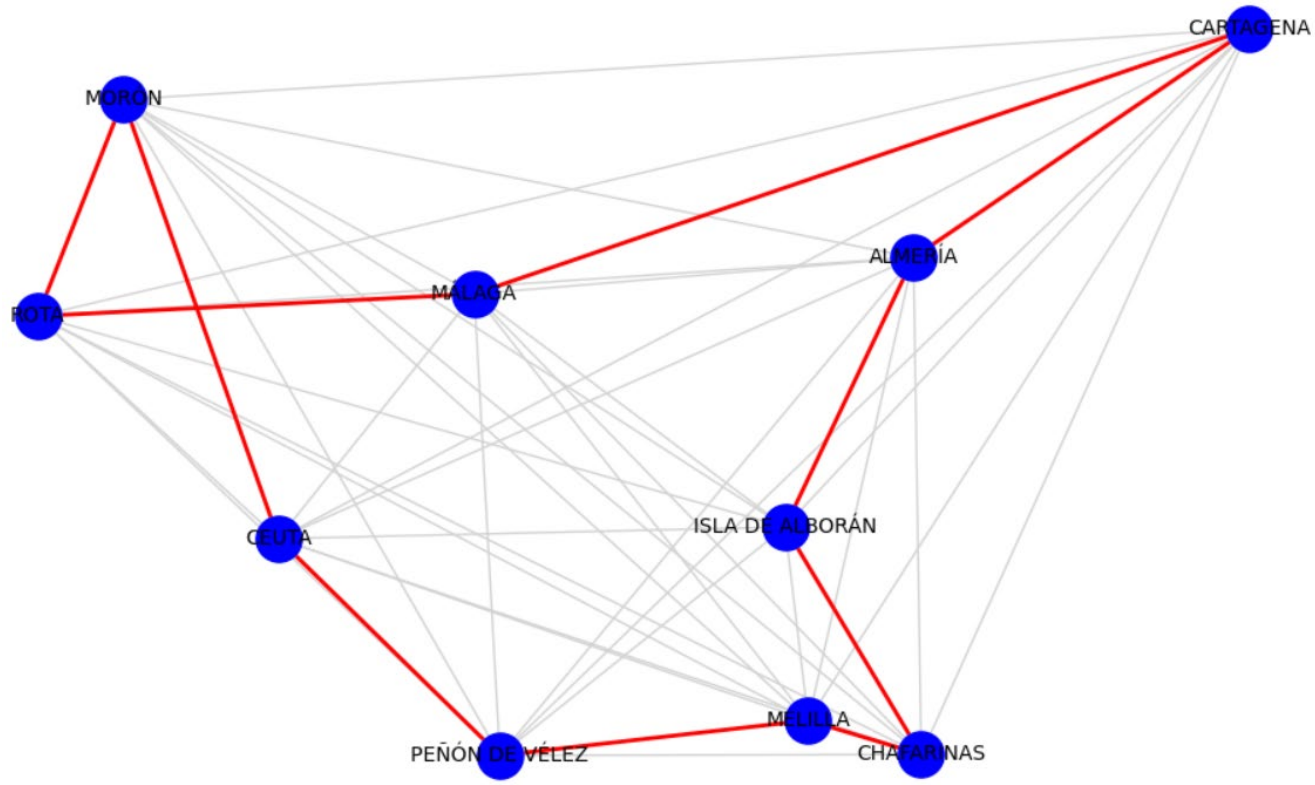

Figura 23. Elaboración propia curso 2025 con programa SageMath. Circuito hamiltoniano solución subóptima mediante método de Christófides.

Este recorrido representa un coste total de $\mathbf{763,36}\boldsymbol{MN}$. Como se puede observar no es la solución óptima, pero se aproxima bastante. Esta solución contiene un error del 4,77% lo cual se encuentra dentro del límite que nos asegura el algoritmo de Christófides que asegura un error máximo del 50%. En este caso se tiene que:

$$w(Christófides) = 1{,}0477 \times w(óptimo)$$

4.3 Resolución por métodos heurísticos

4.3.1 Cota superior. Método de inserción más cercana.

En este apartado se presenta la implementación del método de inserción más cercana, una heurística utilizada para obtener una solución aproximada a este caso práctico. La idea fundamental de este método consiste en comenzar con un ciclo inicial formado por un nodo de partida; en este caso, $Morón$, y la ciudad más próxima a este; dígase, $Rota$, y luego expandir el ciclo de manera iterativa insertando, en cada paso, el nodo que minimice la distancia adicional requerida para ser incorporado.

Para ilustrar este proceso, y facilitar la comprensión del algoritmo, se ha generado mediante el programa SageMath una representación gráfica de cada una de las iteraciones.

En primer lugar, se tiene que $Rota$ es el nodo más cercano a $Morón$, por lo que se puede concluir que este será el ciclo inicial de partida:

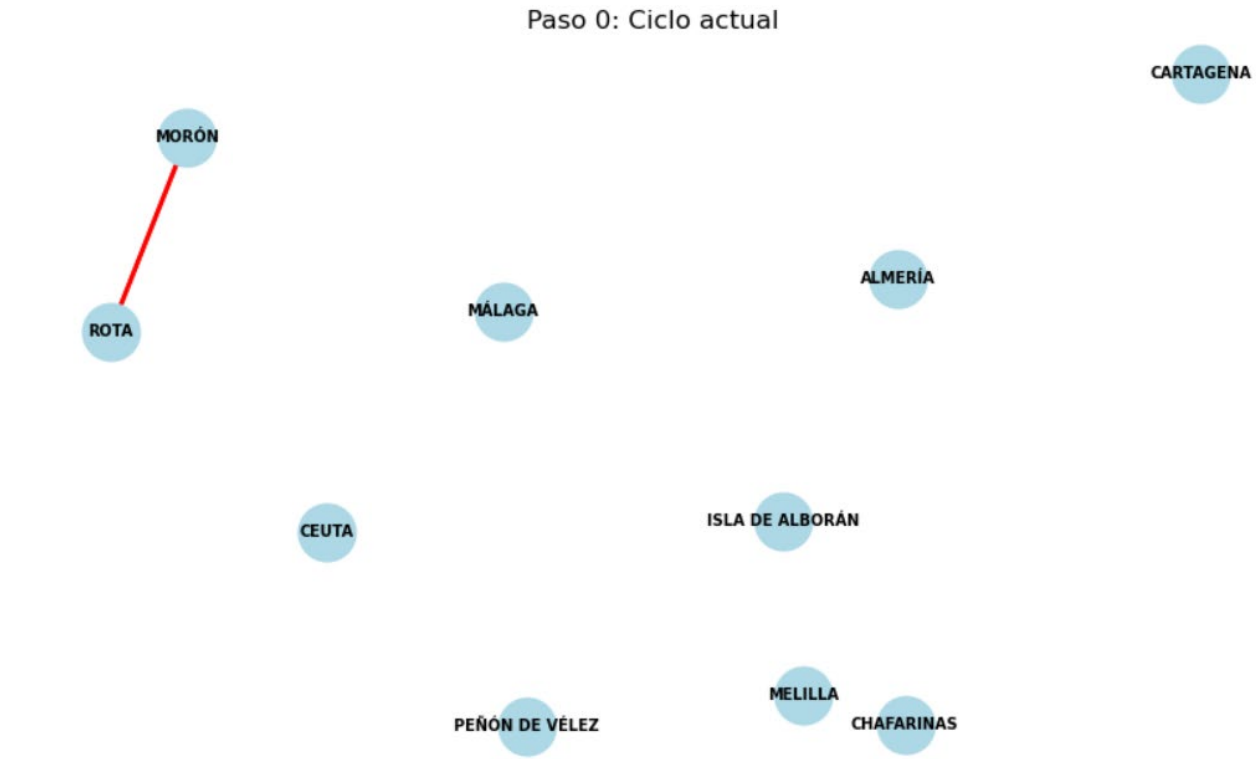

Figura 24. Elaboración propia curso 2025 a través de SageMath. Ciclo de partida para el método de inserción más cercana.

Nota 4.3.1.1. La arista de este ciclo inicial es doble, creando así un ciclo cerrado con dos nodos.

El nodo más cercano a alguno de los dos puntos es $Ceuta$, por lo que se inserta en el ciclo en la posición adecuada:

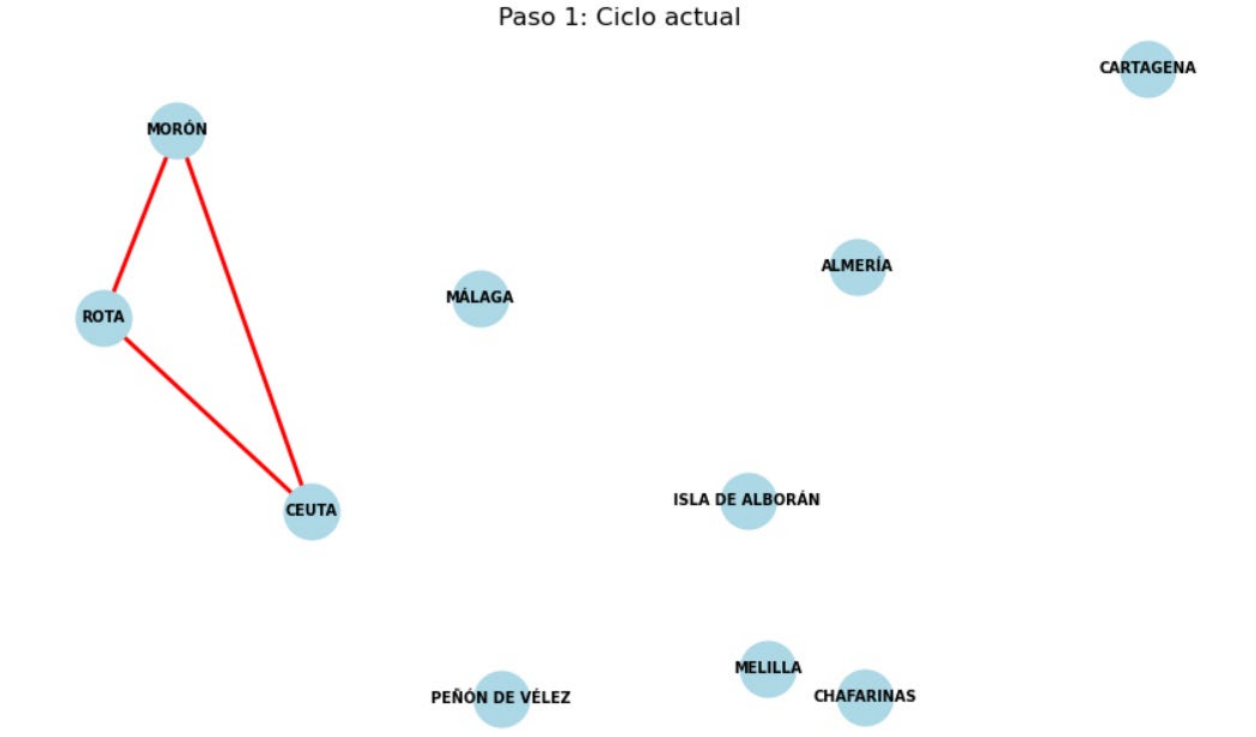

Figura 25. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de $\boldsymbol{Ceuta}$.

A continuación, por su cercanía a $Ceuta$, se inserta $Málaga$ en la posición correspondiente:

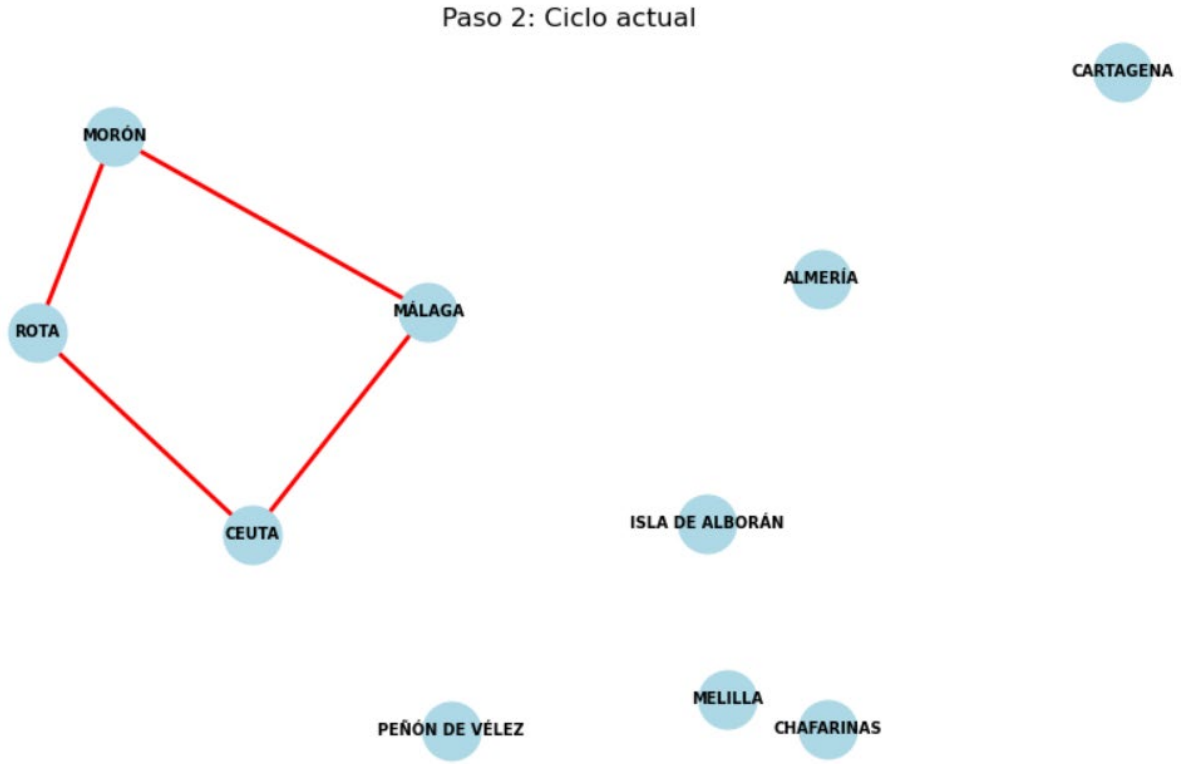

Figura 26. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Málaga.

De nuevo por su cercanía a $Ceuta$, se inserta $Peñón\ de\ Vélez$ en su correspondiente posición:

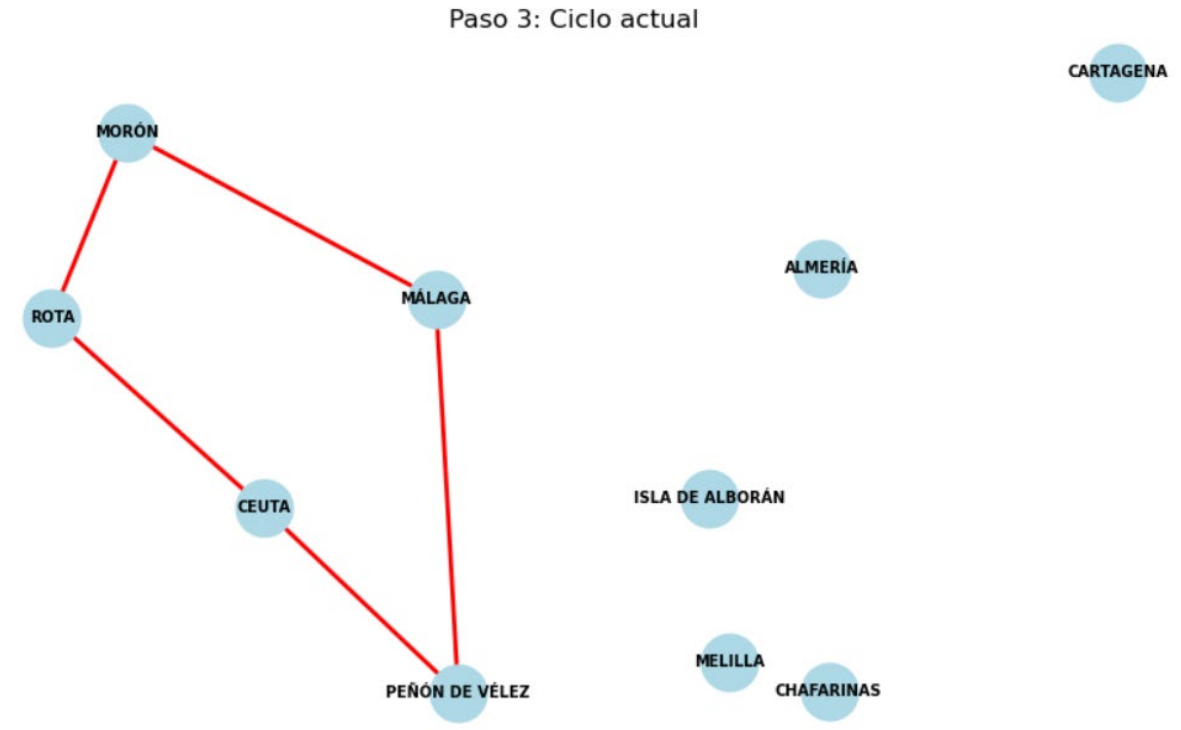

Figura 27. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de Peñón de Vélez.

Por la cercanía al $Peñón\ de\ Vélez$, se inserta detrás, $Melilla$:

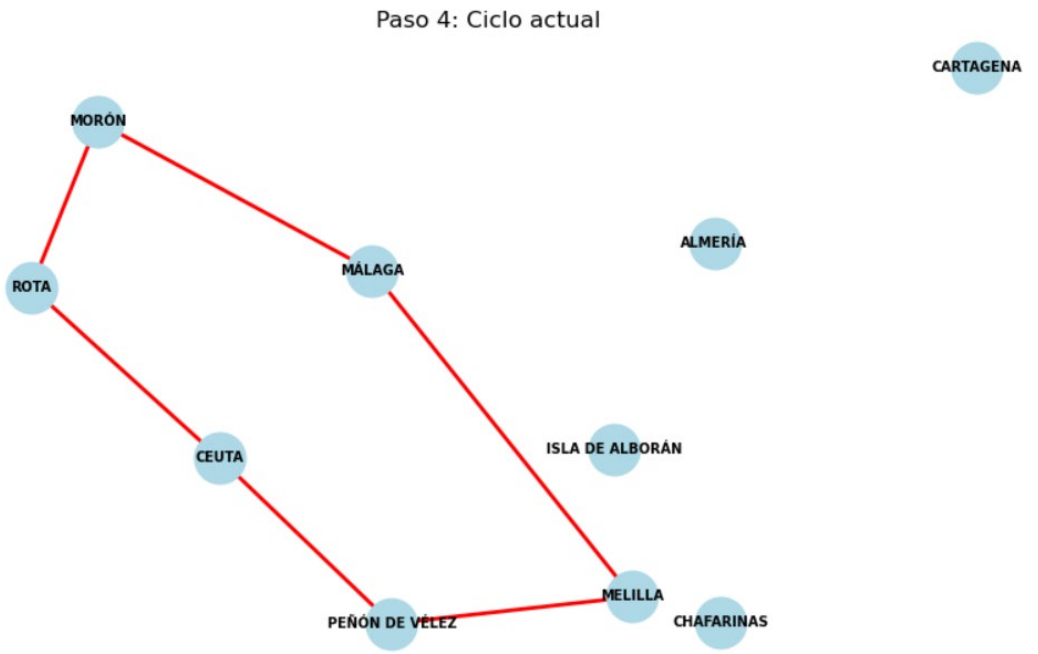

Figura 28. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de $\boldsymbol{Melilla}$.

Por la cercanía a $Melilla$, se inserta detrás $Chafarinas$:

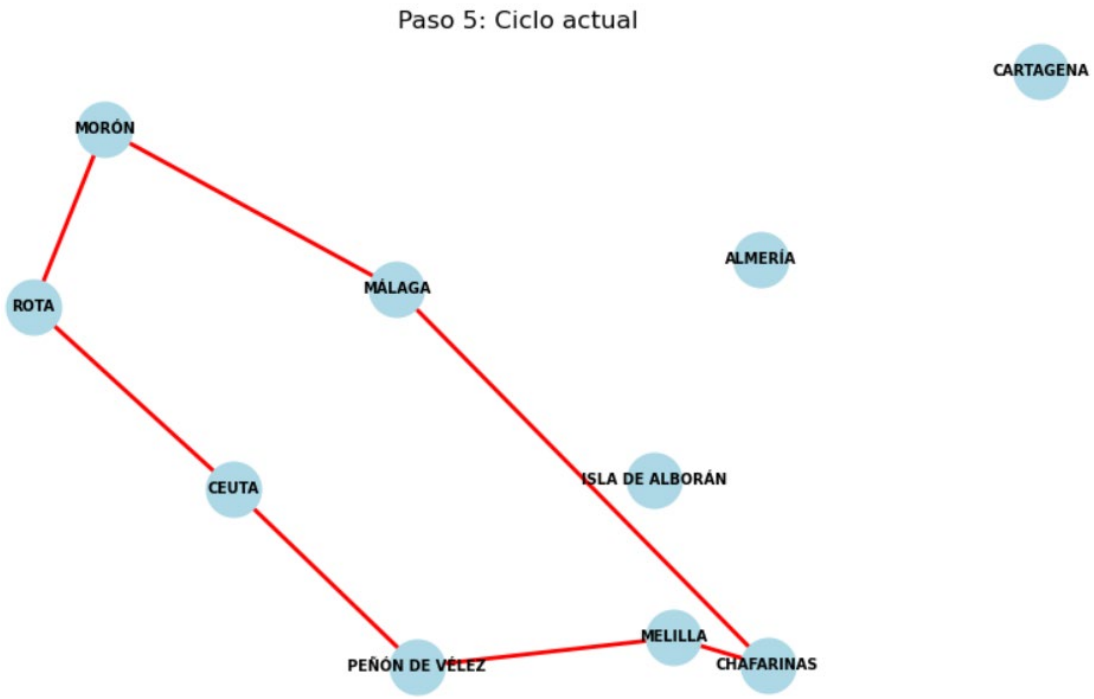

Figura 29. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de $\boldsymbol{Chafarinas}$.

Por la cercanía a $Melilla$ de nuevo, se inserta detrás $Isla\ de\ Albor\acute{a}n$, quedando esta así entre $Melilla$ y $Chafarinas$:

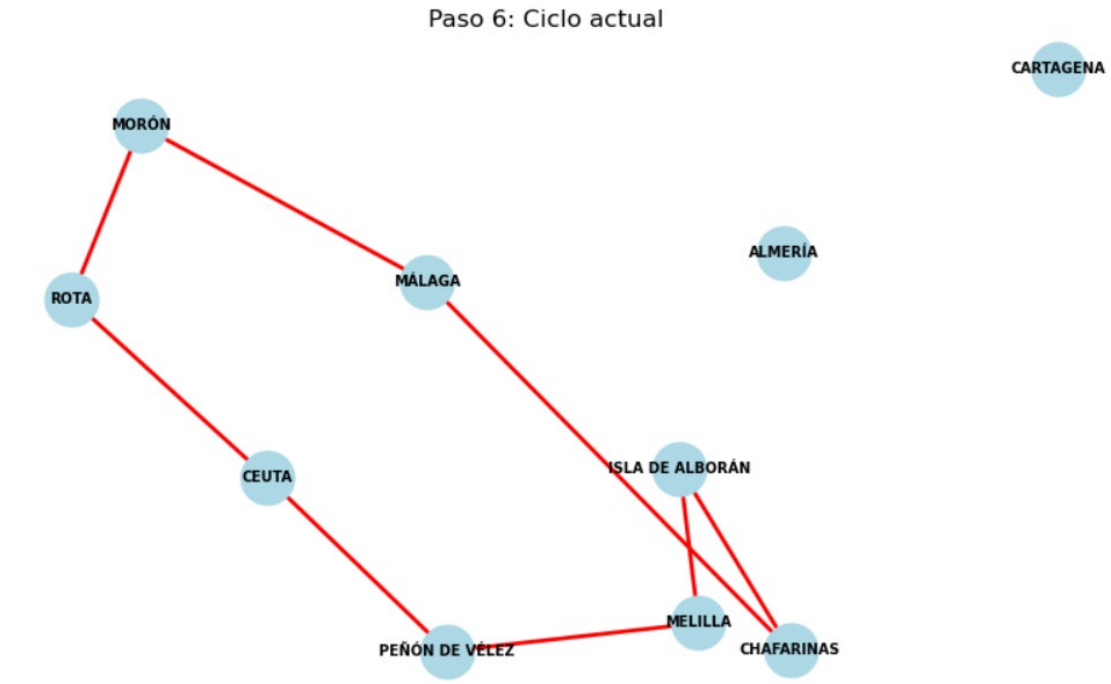

Figura 30. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de ***Isla de Alborán***.

Por la cercanía a $Isla\ de\ Albor\acute{a}n$, se inserta detrás $Almería$:

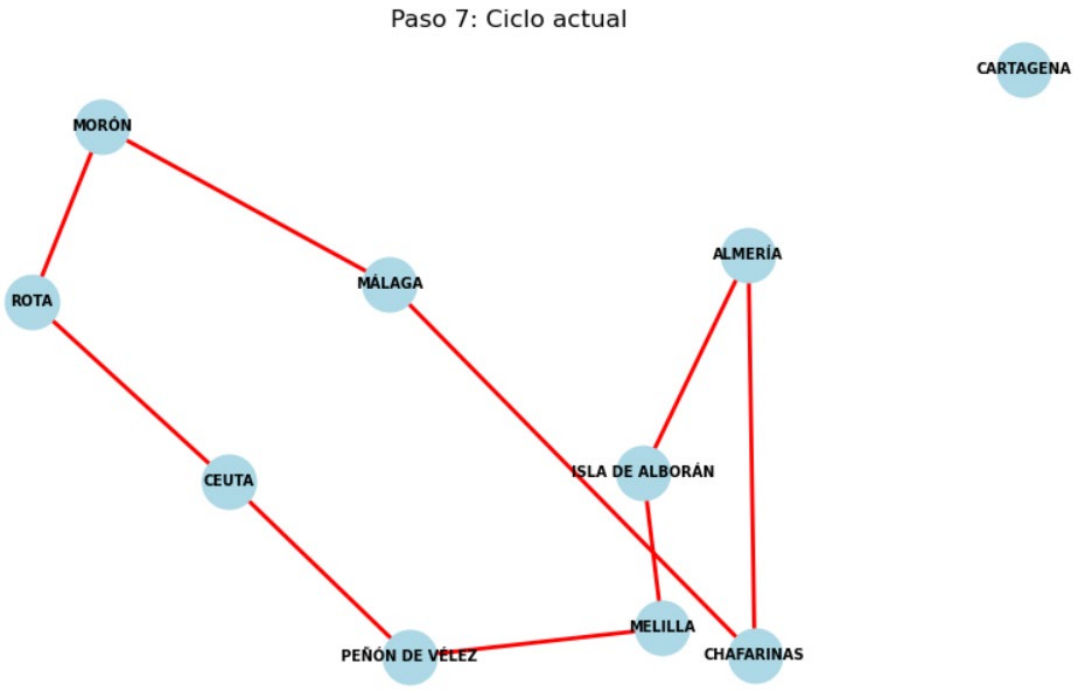

Figura 31. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de ***Almería***.

Por último, por la cercanía con $Almería$, se inserta detrás $Cartagena$. De esta forma, se involucran la totalidad de los nodos, quedando de esta forma definido el ciclo Hamiltoniano completo que establezca una cota superior para la resolución del problema:

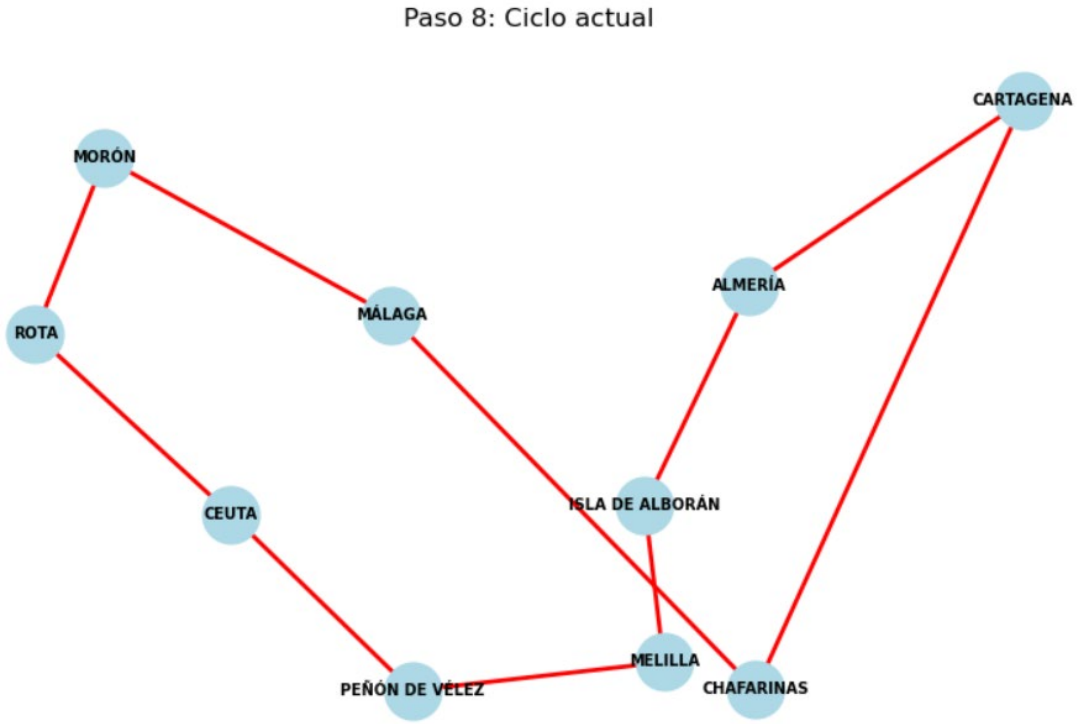

Figura 32. Elaboración propia curso 2025 a través de SageMath. Ciclo tras la inserción de $\boldsymbol{Cartagena}$.

El ciclo resultante es el siguiente:

$$Morón \rightarrow Rota \rightarrow Ceuta \rightarrow Peñón\ de\ Vélez \rightarrow Melilla \rightarrow Isla\ de\ Alborán \rightarrow Almería \rightarrow Cartagena \rightarrow Chafarinas \rightarrow Málaga \rightarrow Morón$$

El coste total de este circuito Hamiltoniano, es una solución válida para nuestro problema TSP, aunque no la óptima como se puede apreciar gráficamente. El coste total del ciclo es de $\boldsymbol{809,03MN}$, lo que delimita una cota superior. Este resultado es $80{,}42MN$ superior al ideal, lo que supone que es un $11{,}04\%$ más largo que el resultado óptimo. Mientras el método heurístico de inserción más cercana puede dar resultados cercanos a la solución óptima en muchos casos; especialmente en problemas de tamaño moderado, en general su coste total suele ser mayor que el óptimo, aunque se compensa con su eficiencia y simplicidad en la implementación.

4.3.2 Método del árbol.

En el problema de la optimización de la ruta de la patrulla aérea, el método del árbol se presenta como una estrategia eficiente para generar un recorrido aproximado al valor real. Este método se basa en la construcción de un MST a partir del grafo completo. Dado que el MST no forma un ciclo, se aplican transformaciones para convertirlo en un recorrido viable para la patrulla. Este método heurístico asegura una solución rápida computacionalmente hablando, aunque no garantiza un porcentaje máximo de error. Sólo se puede asegurar que el resultado final no va a ser mayor al doble del resultado óptimo.[2]

Expuesto esto; en la práctica, la solución obtenida suele ser mejor que este límite teórico, dependiendo de la distribución de los puntos y la naturaleza del problema. Pese a no asegurar la proximidad, estadísticamente suele dar resultados razonablemente buenos. [23]

1) En primer lugar, se genera el árbol de mínima expansión que ya se calculó para el método de Christófides. Figura 19.

2) A continuación, se duplican todas las aristas para obtener un ciclo euleriano:

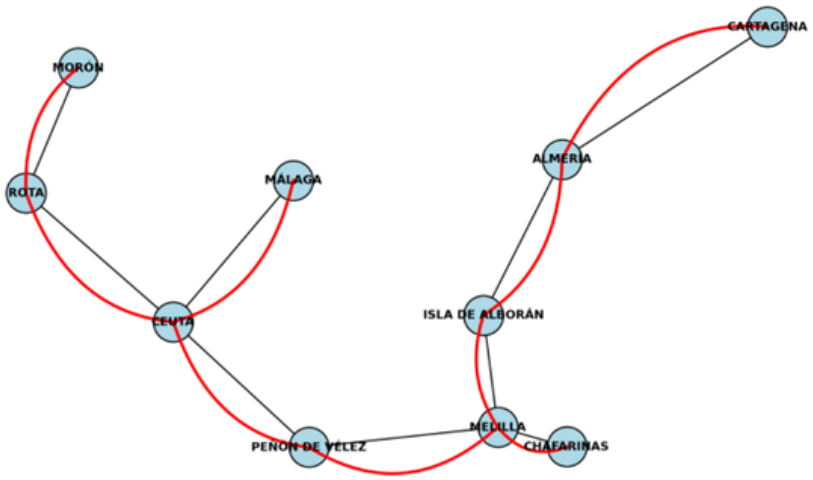

Figura 33. Elaboración propia curso 2025 a través de SageMath con ayuda de IA generativa. La IA se ha empleado para generar la línea roja curva que simula la duplicidad de la arista del MST para favorecer la visualización. Ciclo euleriano con aristas duplicadas.

3) A raíz de este grafo euleriano se crea el circuito euleriano correspondiente, mediante el método de Hierholzer. En el ejercicio, el ciclo recorre una única vez cada una de las aristas de la figura 33 y al menos dos veces cada uno de los vértices, empezando desde $Morón$. El ciclo Euleriano sería el siguiente:

$$Morón \rightarrow Rota \rightarrow Ceuta \rightarrow Peñón\ de\ Vélez \rightarrow Melilla \rightarrow Chafarinas \rightarrow Melilla \rightarrow Isla\ de\ Alborán \rightarrow Almería \rightarrow Cartagena \rightarrow Almería \rightarrow Isla\ de\ Alborán \rightarrow Melilla \rightarrow Peñón\ de\ Vélez \rightarrow Ceuta \rightarrow Málaga \rightarrow Ceuta \rightarrow Rota \rightarrow Morón$$

4) Mediante la eliminación de nodos repetidos se genera un ciclo Hamiltoniano a partir del ciclo Euleriano expuesto. En este caso, el ciclo obtenido es el siguiente:

$$Morón \rightarrow Rota \rightarrow Ceuta \rightarrow Peñón\ de\ Vélez \rightarrow Melilla \rightarrow Chafarinas \rightarrow Isla\ de\ Alborán \rightarrow Almería \rightarrow Cartagena \rightarrow Málaga \rightarrow Morón$$

Podemos ver qué, en este caso, el resultado coincide con la solución óptima, $728{,}61MN$. Como ya se ha analizado, esto no tendría por qué ser así, pues el método del árbol en lo absoluto asegura una solución óptima. La lectura que se puede hacer de la obtención de este resultado es que este método con una disposición de puntos no demasiado densa y en problemas no demasiado grandes; es decir, en problemas como el nuestro, el resultado tiende a ser bastante cercano al óptimo. Aun así, no es posible asegurar que esto vaya a ser así, por lo que sería de utilidad contrastar un resultado obtenido por este método, con el de algún otro de los métodos de los que se ha hablado en este trabajo.

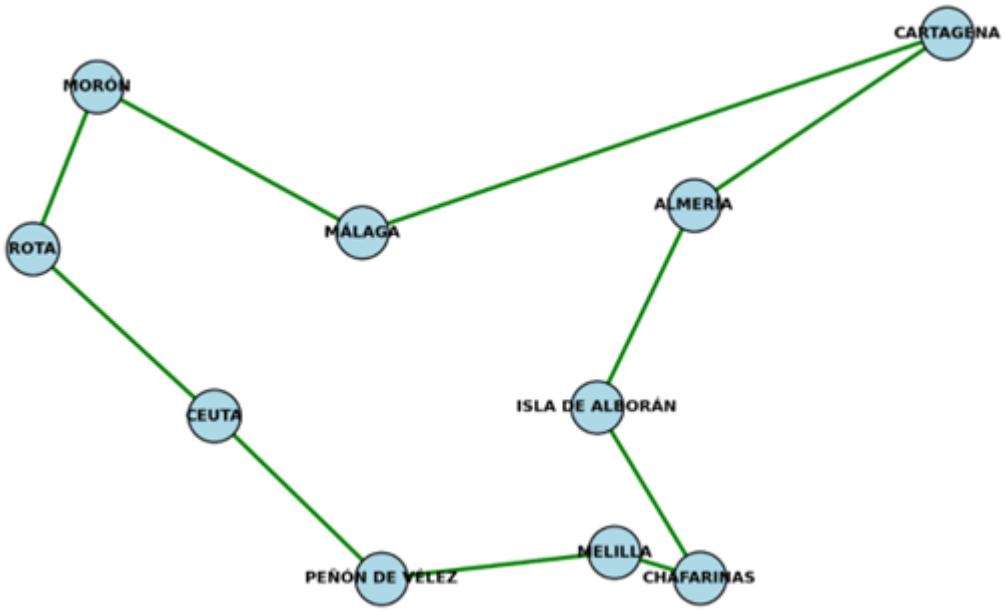

Figura 34. Elaboración propia curso 2025 con programa SageMath. Ciclo hamiltoniano obtenido con el ***método del árbol***.

5 CONCLUSIONES Y LÍNEAS FUTURAS

La comparación de los distintos métodos de resolución del Problema del Viajante de Comercio realizada en este trabajo revela una correlación clara entre el coste computacional y la calidad de la solución obtenida. En general, a mayor esfuerzo de cómputo, mejor es la aproximación al óptimo, mientras que los métodos más eficientes en tiempo suelen implicar alguna pérdida de calidad en la solución. El método de fuerza bruta ejemplifica el caso más extremo de alto coste y a la vez alta calidad; garantiza encontrar la ruta óptima exacta, pero su complejidad exponencial lo vuelve impracticable excepto para casuísticas muy pequeñas. Por su parte, el método del 1-árbol proporciona de forma relativamente eficiente una cota inferior para la longitud mínima del recorrido; es decir, un valor menor que el óptimo, pero que ayuda a delimitar la solución. Esta aproximación, emplea árboles de expansión mínima y un nodo con grado dos para estimar el mejor caso posible. Si bien es útil para acotar el problema y se emplea en técnicas exactas como la ramificación y acotación, el 1-árbol no genera directamente un ciclo hamiltoniano válido. Además, incorporarlo en un algoritmo exacto sigue llevando a un crecimiento exponencial en el peor caso, aunque reduce drásticamente el tiempo de búsqueda comparado con la fuerza bruta. En contraste con los enfoques anteriores, los métodos aproximados y heurísticos ofrecen soluciones de razonable buena calidad, con un coste computacional mucho menor, asumiendo cierta degradación en la optimalidad. El algoritmo de Christófides, es especialmente notable. Basado en combinar árboles de expansión mínima, emparejamientos mínimos y recorridos eulerianos, es capaz de generar un ciclo cuya longitud no supera en más de un 50% a la óptima en problemas métricos. En general, brinda un buen compromiso entre calidad y tiempo de cálculo, resolviendo problemas de tamaño moderado en tiempo polinómico. Por otro lado, la heurística de inserción más cercana construye rápidamente una ruta factible; es decir, ciclo hamiltoniano, actuando de esta forma como cota superior del óptimo. Su principal ventaja es la simplicidad y rapidez de ejecución, lo que permite obtener en segundos una ruta razonable incluso para problemas de mayor tamaño. No obstante, las rutas obtenidas por inserción más cercana suelen ser más largas que la mínima posible; es decir, se sacrifica precisión a cambio de eficiencia, y no existe una garantía teórica acotada del error como en Christófides. De modo similar, el método basado en árboles de expansión mínima; es decir, doblando aristas del árbol para formar un ciclo, mostró ser muy rápido y sencillo de implementar. En nuestro caso de estudio; en el punto 4.3.2, este método del árbol produjo la ruta óptima, debido a la dispersión de los puntos y a que no resulta un ejemplo demasiado complejo. En general, este método solo asegura soluciones cuya longitud puede ser hasta el doble de la óptima en el peor de los casos. Aun así, su baja complejidad lo convierte en una herramienta valiosa cuando prima la rapidez sobre la exactitud. Concluyendo, los resultados ponen de manifiesto que existe un equilibrio notable entre el tiempo de computación y la calidad de la solución en el TSP. Los métodos exactos garantizan la mejor solución posible a cambio de asumir un coste computacional prohibitivo. Por otra parte, las aproximaciones heurísticas logran soluciones relativamente cercanas al óptimo en tiempos razonables. La elección del enfoque más adecuado, dependerá del tamaño del problema y de las necesidades concretas, dependiendo si se requiere la solución perfecta o si basta con una buena solución obtenida con rapidez.

A la pregunta de si algún día se encontrará una solución óptima para el TSP en tiempo polinomial, la mayoría de la comunidad científica cree que no, al menos bajo el paradigma actual, pues aceptar que el TSP es resoluble en tiempo polinómico implicaría que P = NP. Esto sentaría una afirmación sin precedentes, siendo este un resultado que colapsaría las fronteras de la informática conocida, y redefiniría nuestra noción de lo computable. Sin embargo, como ocurre con muchas grandes preguntas de la ciencia, la esperanza permanece, alimentada por la creatividad, los avances tecnológicos y la posibilidad de que aún no hayamos alcanzado la perspectiva adecuada para ver una solución que siempre estuvo ahí.

A partir de este trabajo se abren diversas líneas futuras de interés tanto práctico como de investigación, entre las cuales cabe destacar:

- Aplicaciones en sistemas militares, tales como la optimización logística y la reducción de costes operativos. Una línea de trabajo futura de interés, sería la aplicación del TSP en el ámbito militar, no solo para planificar rutas de patrullaje como se ha realizado en el presente trabajo, sino para optimizar los procesos logísticos y de aprovisionamiento en operaciones terrestres o navales. Por ejemplo, en escenarios de despliegue de unidades en misiones internacionales o ejercicios multinacionales, se requiere transportar suministros cómo pudieran ser, munición, combustible, víveres o repuestos, entre múltiples puntos geográficos con recursos limitados. En estos casos, el uso de algoritmos derivados del TSP podría permitir diseñar rutas de distribución más eficientes para vehículos terrestres o embarcaciones auxiliares, minimizando las millas a recorrer, el consumo de combustible y el desgaste del material, lo cual se traduce directamente en una reducción de costes operativos. Esto puede resultar especialmente relevante en misiones prolongadas o con restricciones logísticas severas, como en zonas de difícil acceso o con presencia limitada de infraestructuras. Asimismo, estas herramientas podrían aplicarse a la planificación de convoyes logísticos, asignación de tareas en bases avanzadas o en la gestión del inventario distribuido, en contextos donde es prioritario realizar el menor número posible de movimientos entre almacenes o unidades desplegadas. El TSP, sin lugar a dudas, podría convertirse en un recurso recurrente para aumentar la eficiencia y sostenibilidad de las operaciones militares, más allá del enfoque táctico o de vigilancia tratado en este TFG.
- Se podría extender el estudio a problemas de enrutamiento derivados. Otra vía de desarrollo es ampliar el estudio a problemas afines más complejos que derivan del TSP. Un ejemplo es el Problema de Enrutamiento de Vehículos o Vehicle Routing Problem, (VRP), donde intervienen múltiples vehículos con capacidad limitada o diferentes puntos de inicio/fin, y el Problema de Recogida y Entrega o Pickup and Delivery Problem, (PDP), que añade restricciones de recoger y dejar mercancías o pasajeros. Estos problemas reflejan mejor las necesidades logísticas reales como reparto de mercancías o rutas de transporte y su resolución suele requerir adaptar o complementar las técnicas del TSP. Explorar cómo los métodos comparados en este trabajo podrían aplicarse o modificarse para resolver VRP o PDP sería de interés, permitiendo abordar casos de estudio más realistas en distribución y transporte.
- Se podrían poner en práctica ciertos métodos metaheurísticos. Sería de interés implementar y evaluar metaheurísticas de última generación para la resolución del TSP y sus variantes, mediante algoritmos evolutivos como los algoritmos genéticos, o técnicas bio-inspiradas como la optimización por colonia de hormigas, que han demostrado en numerosos estudios su capacidad para encontrar soluciones cercanas al óptimo en problemas $NP-hard$ con un esfuerzo computacional moderado. La aplicación de estas metaheurísticas a un caso de estudio permitiría manejar problemas de mayor tamaño o complejidad, a la vez que potencialmente mejora la calidad de las soluciones obtenidas por heurísticas simples. Desarrollar e incorporar estos enfoques avanzados podría ser una extensión lógica para lograr una visión más vanguardista sobre el estado actual de los avances del TSP.

6 BIBLIOGRAFÍA

[1] Applegate, D. L. (2006). The traveling salesman problem: a computational study (Vol. 17). Princeton university press.

[2] Graphs, Networks and Algorithms (4th ed.) [Jungnickel 2012-11-09]

[3] Aldous, Joan M., Wilson, Robin J. Graphs and Applications. Springer, 2006.

[4] Held, M., and Karp, R. A dinamic programming approach to sequencing problems. SIAM J. Appl. Math. 10, 1962.

[5] Held, M., and Karp, R. The traveling salesman problem and minimum spanning trees. Oper. Res. 18, 1970.

[6] Held, M., and Karp, R. The traveling salesman problem and minimum spanning trees: part II. Math. Programming I, 1971

[7] Wells, Jack (19 de marzo de 2018). «Powering the Road to National HPC Leadership». OpenPOWER Summit 2018.

[8] Nguyen, Q. N. (2020). Traveling Salesman Problem and Bellman-Held-Karp Algorithm.

[9] Tito Ontaneda, J. E., & Yacelga Pinto, M. E. (2018). Comparación de un método exacto y aproximado en la resolución del TSP para una WSN.

[10] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, 7(1), 48–50.

[11] Gauthier, J., Vincent, A. T., Charette, S. J., & Derome, N. (2019). A brief history of bioinformatics. Briefings in bioinformatics, 20(6), 1981-1996.

[12] Christofides, N. (1976). The vehicle routing problem. Revue française d'automatique, informatique, recherche opérationnelle. Recherche opérationnelle, 10(V1), 55-70.

[13] Ackermann, W. (1928). Zum hilbertschen aufbau der reellen zahlen. Mathematische Annalen, 99(1), 118-133.

[14] Bergantiños, G., & Vidal-Puga, J. (2011). The folk solution and Boruvka's algorithm in minimum cost spanning tree problems. Discrete applied mathematics, 159(12), 1279-1283.

[15] SILVA, A. S. (2015). O Teorema de Euler e Algumas Aplicações. TCC apresentado a Universidade Estadual da Paraiba, Campina Grande.

[16] Hincapié, R. A., Ríos Porras, C. A., & Gallego, R. A.(2004) "Técnicas heurísticas aplicadas al problema del cartero viajante", Scientia et Technica, Universidad Tecnológica de Pereira.

[17] Gottlieb, E. (2005). La fórmula de Euler: poliedros, grafos planares y topología. Revista del instituto de matemática y física.

[18] Hierholzer, C., & Wiener, C. (1873). Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. Mathematische Annalen, 6(1), 30-32.

[19] «El mar de Alborán, cementerio de inmigrantes: las autoridades recuperaron 67 cadáveres en 2023». Accedido: 11 de marzo de 2025. [En línea]. Disponible en: https://www.eldebate.com/espana/andalucia/20240419/mar-alboran-cementerio-inmigrantes-autoridades-recuperaron-67-cadaveres-2023_190463.html

[20] «¿Cómo calcular la distancia entre dos puntos geográficos en C#? (Fórmula de Haversine)». Accedido: 11 de marzo de 2025. [En línea]. Disponible en: https://www.genbeta.com/desarrollo/como-calcular-la-distancia-entre-dos-puntos-geograficos-en-c-formula-de-haversine

[21] Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman

[22] Bläser, Markus (2008), "Metric TSP", in Kao, Ming-Yang (ed.), Encyclopedia of Algorithms}, Springer-Verlag, pp. 517–519, ISBN 9780387307701.

[23] Cunquero, R. M. (2003). Algoritmos heurísticos en optimización combinatoria. Valencia: Universodad de Valencia. Retrieved, 11(01), 2012.

[24] Apple, K., & Haken, W. (2009). Siete puentes, un camino: Königsberg. Revista suma, 45, 69-78.

[25] «El problema de los puentes de Königsberg». Accedido: 17 de marzo de 2025. [En línea]. Disponible en: https://ingenieriabasica.es/el-problema-de-los-puentes-de-konigsberg/

[26] Hamilton, W. R. (2006). The Mathematical Papers of Sir William Rowan Hamilton. CUP Archive.

[27] Sahalot, A., & Shrimali, S. (2014). A comparative study of brute force method, nearest neighbour and greedy algorithms to solve the travelling salesman problem. International Journal of Research in Engineering & Technology, 2(6), 59-72.

[28] Bean, D. R. (1976). Recursive Euler and Hamilton paths. Proceedings of the American Mathematical Society, 55(2), 385-394.

[29] Ball, W. W. R. and Coxeter, H. S. M. Mathematical Recreations and Essays, 13th ed. New York: Dover, pp. 262-266, 1987.

[30] Sigmund, K. (1998). Menger's Ergebnisse—a biographical introduction. Karl Menger: Ergebnisse eines Mathematischen Kolloquiums, 5-31.

[31] Flood, M. M. (1956). The traveling-salesman problem. Operations research, 4(1), 61-75.

[32] Whitney, H. (1931). Non-separable and planar graphs. Proceedings of the National Academy of Sciences, 17(2), 125-127.

[33] Miller, C. E., Tucker, A. W., & Zemlin, R. A. (1960). Integer programming formulation of traveling salesman problems. Journal of the ACM (JACM), 7(4), 326-329.

[34] Dantzig, G. B. (1990). Origins of the simplex method. In A history of scientific computing (pp. 141-151).

[35] Dantzig, G., Fulkerson, R., & Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. Journal of the operations research society of America, 2(4), 393-410.

[36] Boyd, S., & Mattingley, J. (2007). Branch and bound methods. Notes for EE364b, Stanford University, 2006, 07.

[37] Little, J. D., Murty, K. G., Sweeney, D. W., & Karel, C. (1963). An algorithm for the traveling salesman problem. Operations research, 11(6), 972-989.

[38] Zambito, L. (2006). The traveling salesman problem: a comprehensive survey. Project for CSE, 4080, 11.

[39] Cook, S. A. (2023). The complexity of theorem-proving procedures. In Logic, automata, and computational complexity: The works of Stephen A. Cook (pp. 143-152).

[40] Karp, R. M. (2009). Reducibility among combinatorial problems. In 50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art (pp. 219-241). Berlin, Heidelberg: Springer Berlin Heidelberg.

[41] Sanches, D., Whitley, D., & Tinós, R. (2017, July). Improving an exact solver for the traveling salesman problem using partition crossover. In Proceedings of the Genetic and Evolutionary Computation Conference (pp. 337-344).

[42] DE MST, F. U. N. D. A. M. E. N. T. O. S. (2002). Algoritmo evolutivo para el problema de árbol de expansión mínima (MST). Industrial Data, 5(2), 64-67.

[43] Moujahid, A., Inza, I., & Larrañaga, P. (2008). Tema 2. algoritmos genéticos. Departamento de Ciencia de la Computación e Inteligencia Artificial Universidad del País Vasco-Euskal Herriko Unibertsitatea, 1-33.

[44] Kirkpatrick, S., Gelatt Jr, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. science, 220(4598), 671-680.

[45] López, E., Salas, Ó., & Murillo, Á. (2014). The traveling salesman problem: a deterministic algorithm using tabu search. Revista de Matemática Teoría y Aplicaciones, 21(1), 127-144.

[46] Alonso, S., Cordón, O., Fernández, I., & Herrera, F. (2004). La metaheurística de optimización basada en colonias de hormigas: modelos y nuevos enfoques. Optimización inteligente: técnicas de inteligencia computacional para optimización, 261-314.

[47] Cárdenas-Haro, J. A., Vargas-Figueroa, A. J., & Maupome-Polanco, A. F. (2013). La Notación Asintótica En El Cómputo Científico. Tlatemoani, (12).

[48] Naval Air Systems Command. JP-5 and JP-8 Aviation Fuels – Military Specifications and Characteristics. NAVAIR 00-110-300, 2020.

[49] IATA Jet Fuel Price Monitor. Spot Jet Fuel Price Index. https://www.iata.org/en/publications/economics/fuel-monitor/

ANEXO I: IMPACTO ECONÓMICO Y AMBIENTAL

La optimización de rutas de patrulla mediante técnicas inspiradas en el Problema del Viajante de Comercio, puede aportar beneficios significativos en términos de reducción de costes operativos y disminución del impacto ambiental. En este anexo se analizan dichas implicaciones, empleando aeronaves AV-8B Harrier II de la Armada Española. Se centra principalmente en el consumo de combustible de estos aviones, el precio del combustible JP-5, la estimación de ahorros potenciales con rutas optimizadas tanto un 10% como un 15% y en cómo la reducción de consumo conlleva beneficios ambientales y logísticos.

Los AV-8B Harrier II; aviones de despegue vertical utilizados por la Armada, son aeronaves de combate monomotor con un consumo de combustible considerable. Cuentan con una capacidad interna de combustible de aproximadamente 4.300 litros. Este combustible empleado es queroseno JP-5. En condiciones de vuelo de crucero eficientes a altitud, un Harrier II puede alcanzar autonomías del orden de 700 millas náuticas con el tanque interno lleno, lo que implica un consumo promedio cercano a 3.000 litros por hora de vuelo; en perfil de traslado o ferry. Sin embargo, en perfiles tácticos, cómo patrullas a baja cota, maniobras, ascensos o descensos, el consumo específico aumenta., aunque no es el caso de misiones de patrullaje.

El JP-5, es el combustible de aviación empleado por las aeronaves embarcadas en la armada española y por lo tanto por los Harrier II. El precio del JP-5 fluctúa con el mercado del petróleo, pero es similar al del jet fuel civil, conocido cómo Jet-A1. A fecha de 2025, su valor se sitúa aproximadamente entre 0,55 y 0,60 € por litro sin tener en cuenta impuestos. Este coste por litro de combustible convierte al consumo de los Harrier II en un factor económico relevante. Una patrulla prolongada que queme, por ejemplo, 5.000 litros de JP-5 representa un gasto de alrededor de 3.000 € directamente en combustible. A ello se suman costes indirectos asociados, cómo desgaste de motores u mantenimiento por hora de vuelo, aunque en este anexo el foco se centra en el coste del combustible como el principal componente variable que la optimización de rutas puede disminuir. [48]

Aplicar métodos de optimización basados en el TSP a la planificación de patrullas aéreas puede reducir la distancia total recorrida hasta un 10 o un 15%, suponiendo que la ruta inicial seleccionada no es la óptima. Una disminución de la distancia recorrida se traduce en ahorro de combustible, asumiendo que el consumo de un reactor en crucero es aproximadamente proporcional a la distancia que dicho avión recorre. En la siguiente tabla se muestra una estimación del consumo de combustible y el ahorro potencial para distintos escenarios de distancia que recorre una patrulla. Se considera en esta tabla un consumo medio de $5l/km$ y un coste de JP5 de $0{,}60€/l$ [49]:

Distancia	Consumo inicial	Ahorro con ruta 10% optimizada	Ahorro con ruta 15% optimizada
539,974MN (1000km)	$5000l{\sim}3000€$	$500l{\sim}300€$	$750l{\sim}450€$
809,935MN (1500km)	$7500l{\sim}4500€$	$750l{\sim}450€$	$1125l{\sim}675€$
1079,91MN (2000km)	$10000l{\sim}6000€$	$1000l{\sim}600€$	$1500l{\sim}900€$

Tabla 7. Consumo de combustible estimado y ahorros asumiendo optimizaciones de la ruta de un 10% y un 15%

Teniendo en cuenta los resultados arrojados en la tabla, se pone de manifiesto un beneficio económico tangible al utilizar métodos de optimización de rutas.

En cuanto al apartado ambiental se refiere, la reducción en el consumo de combustible tiene un efecto directo en el impacto ambiental de las operaciones. El queroseno JP-5, al quemarse, emite una cantidad significativa de CO_2 y otros gases de efecto invernadero. En términos aproximados, la

combustión de 1 litro de queroseno genera del orden de 2,58 kg de CO_2. Por tanto, si por ejemplo una optimización de rutas ahorra 500 L de combustible en una patrulla, eso supone evitar la emisión de aproximadamente 1,3 toneladas de CO_2 a la atmósfera. La optimización en las rutas también contribuye al medio ambiente, generando una menor huella de carbono y reduciendo proporcionalmente otras emisiones contaminantes asociadas a la combustión aeronáutica como los NO_x. Además, desde el punto de vista logístico, un menor consumo implica menos necesidad de reabastecimiento si se toman muestras a largo plazo. Cada litro de JP-5 ahorrado es un litro menos que debe ser transportado hasta el área de operaciones o, por ejemplo, almacenado en el buque portaaeronaves. En operaciones embarcadas, una disminución del consumo prolonga la autonomía de combustible del buque y puede espaciar las maniobras de reaprovisionamiento en la mar por parte de buques de aprovisionamiento de combate como el Patiño o el Cantabria. Igualmente, en operaciones desde bases en tierra, reduce la frecuencia de abastecimiento mediante camiones cisterna u otros medios, aliviando la carga logística. Concluyendo, optimizar rutas de patrullas aéreas para consumir menos combustible aporta; a parte de una ventaja económica y ambiental, una mayor eficiencia operativa si se implementa con vista a medio y largo plazo.

ANEXO II: HERRAMIENTA SAGEMATH PARA APOYAR RESOLUCIÓN

En este anexo se introducirá el código empleado como apoyo en la resolución de este trabajo.

En primer lugar, el código para generar la matriz de distancias a partir de las coordenadas geográficas de los nodos es el siguiente:

```
# Recargar librerías
import numpy as np
import pandas as pd
from geopy.distance import geodesic
from itertools import permutations
# Definir los datos de latitud y longitud
locations = {
    "MORÓN": (37.3597, -5.96755),
    "ROTA": (36.639475, -6.346078),
    "CEUTA": (35.89902, -5.28055),
    "MÁLAGA": (36.7131, -4.4128),
    "PEÑÓN DE VÉLEZ": (35.17785, -4.29972),
    "ISLA DE ALBORÁN": (35.937725, -3.0365),
    "MELILLA": (35.29444, -2.9336),
    "CHAFARINAS": (35.1809, -2.433905),
    "CARTAGENA": (37.595451, -0.98128),
    "ALMERÍA": (36.83433, -2.46785),
}
city_names = list(locations.keys())
coordinates = list(locations.values())
# Calcular la matriz de distancias usando la distancia geodésica (haversine)
distance_matrix = np.zeros((len(city_names), len(city_names)))
for i in range(len(city_names)):
    for j in range(len(city_names)):
        if i != j:
            distance_matrix[i][j] = geodesic(coordinates[i],
coordinates[j]).kilometers
df_distance = pd.DataFrame(distance_matrix)
# Convertir la matriz de distancias a millas náuticas
df_distance_nm = df_distance / 1.852
# Mostrar matriz resultante
print(df_distance_nm)
```

- Lo que entrega el programa al ejecutar este código es la siguiente matriz:

```
             0           1           2           3           4           5  \
0     0.000000   46.835893   93.599870   84.139922  153.738730  165.175807
1    46.835893    0.000000   68.123300   93.413140  132.720747  165.940741
2    93.599870   68.123300    0.000000   64.420228   64.600202  109.385092
3    84.139922   93.413140   64.420228    0.000000   92.145521   81.306207
4   153.738730  132.720747   64.600202   92.145521    0.000000   76.788237
5   165.175807  165.940741  109.385092   81.306207   76.788237    0.000000
6   192.200476  184.693241  120.411487  111.396279   67.506030   38.866063
7   215.459146  209.724884  145.869254  133.113598   91.766818   54.091855
8   238.535687  263.703357  230.861494  172.865044  216.369092  140.290905
9   170.903857  187.388500  147.361324   94.034698  133.418106   60.373284

             6           7           8           9
0   192.200476  215.459146  238.535687  170.903857
1   184.693241  209.724884  263.703357  187.388500
2   120.411487  145.869254  230.861494  147.361324
3   111.396279  133.113598  172.865044   94.034698
4    67.506030   91.766818  216.369092  133.418106
5    38.866063   54.091855  140.290905   60.373284
6     0.000000   25.483789  167.144379   95.001249
7    25.483789    0.000000  160.873803   99.075989
8   167.144379  160.873803    0.000000   84.593647
9    95.001249   99.075989   84.593647    0.000000
```

A continuación, se muestra el código para crear el grafo en el que se muestran todos los nodos unidos entre sí:

```python
#Importar librerías
import matplotlib.pyplot as plt
import numpy as np
import networkx as nx
# Definir los datos de latitud y longitud
locations = {
    "MORÓN": (37.3597, -5.96755),
    "ROTA": (36.639475, -6.346078),
    "CEUTA": (35.89902, -5.28055),
    "MÁLAGA": (36.7131, -4.4128),
    "PEÑÓN DE VÉLEZ": (35.17785, -4.29972),
    "ISLA DE ALBORÁN": (35.937725, -3.0365),
    "MELILLA": (35.29444, -2.9336),
    "CHAFARINAS": (35.1809, -2.433905),
    "CARTAGENA": (37.595451, -0.98128),
    "ALMERÍA": (36.83433, -2.46785),
}
# Crear el grafo
G = nx.Graph()
# Agregar nodos
for city, (lat, lon) in locations.items():
    G.add_node(city, pos=(lon, lat))
# Conectar todos los nodos entre sí
for city1 in locations:
    for city2 in locations:
        if city1 != city2:
```

```
            G.add_edge(city1, city2)
# Obtener posiciones
pos = nx.get_node_attributes(G, 'pos')
# Dibujar el grafo
plt.figure(figsize=(10, 6))
nx.draw(G, pos, with_labels=True, node_size=500, node_color="lightblue",
edge_color="gray", font_size=8, font_weight="bold")
#Añadir datos complementarios de latitud y longitud
plt.xlabel("Longitud (Oeste)")
plt.ylabel("Latitud (Norte)")
plt.title("Conexión entre todas las ubicaciones")
plt.show()
```

El programa al ejecutarse, se muestra el grafo expuesto en la figura 15.

Para la resolución del problema por fuerza bruta se ha empleado este código:

```
# Resolver el TSP con Fuerza Bruta debido al tamaño pequeño del problema
def tsp_brute_force(distance_matrix):
    n = len(distance_matrix)
    min_path = None
    min_cost = float("inf")
    for perm in permutations(range(1, n)):  # Permutaciones excluyendo el nodo de
inicio (0)
        path = (0,) + perm + (0,)  # Volver al inicio
        cost = sum(distance_matrix[path[i]][path[i+1]] for i in range(n))
        if cost < min_cost:
            min_cost = cost
            min_path = path
    return min_path, min_cost
# Ejecutar el algoritmo de fuerza bruta
optimal_path, optimal_cost = tsp_brute_force(df_distance_nm.values)
# Convertir el camino óptimo a nombres de ciudades
optimal_path_cities = [city_names[i] for i in optimal_path]
# Mostrar la matriz de distancias y el resultado óptimo
df_distances = pd.DataFrame(distance_matrix, index=city_names, columns=city_names)
optimal_path_cities, optimal_cost
```

Este código se ejecuta detrás de definir la matriz de distancias arriba expuesta. Ejecutando en el programa ambos códigos juntos (saltándose el paso de mostrar la matriz resultante), el resultado que se devuelve es el siguiente:

```
(['MORÓN',
 'MÁLAGA',
 'CARTAGENA',
 'ALMERÍA',
 'ISLA DE ALBORÁN',
 'CHAFARINAS',
 'MELILLA',
 'PEÑÓN DE VÉLEZ',
 'CEUTA',
 'ROTA',
 'MORÓN'],
728.612967052319)
```

Para generar la gráfica de dicho circuito hamiltoniano, se ha añadido al código anterior las siguientes líneas:

```
import matplotlib.pyplot as plt
# Extraer las coordenadas en el orden del ciclo óptimo
optimal_coords = [coordinates[i] for i in optimal_path]
# Agregar el primer punto para cerrar el ciclo
optimal_coords.append(optimal_coords[0])
# Separar latitudes y longitudes (x: longitud, y: latitud)
lats = [coord[0] for coord in optimal_coords]
lons = [coord[1] for coord in optimal_coords]
# Configurar y dibujar el gráfico
plt.figure(figsize=(8, 6))
plt.plot(lons, lats, marker='o', linestyle='-', color='b')
# Añadir etiquetas a cada punto
for idx in optimal_path:
    plt.text(coordinates[idx][1], coordinates[idx][0], city_names[idx],
             fontsize=9, ha='right', va='bottom')
# Etiqueta para el punto final que es el mismo que el inicial.
plt.text(optimal_coords[-1][1], optimal_coords[-1][0], city_names[optimal_path[0]],
         fontsize=9, ha='right', va='bottom')
plt.xlabel("Longitud")
plt.ylabel("Latitud")
plt.title("Ruta Óptima")
plt.grid(True)
plt.show()
```

Al ejecutarse este código, el programa muestra la imagen correspondiente a la figura 16.

Para la resolución del problema por el método del 1-árbol de manera que se realice por pasos el código empleado es el siguiente:

```
#Recargar librerías
import numpy as np
import pandas as pd
from geopy.distance import geodesic
# 1. Definir los datos de latitud y longitud
locations = {
    "MORÓN": (37.3597, -5.96755),
    "ROTA": (36.639475, -6.346078),
    "CEUTA": (35.89902, -5.28055),
    "MÁLAGA": (36.7131, -4.4128),
    "PEÑÓN DE VÉLEZ": (35.17785, -4.29972),
    "ISLA DE ALBORÁN": (35.937725, -3.0365),
    "MELILLA": (35.29444, -2.9336),
    "CHAFARINAS": (35.1809, -2.433905),
    "CARTAGENA": (37.595451, -0.98128),
    "ALMERÍA": (36.83433, -2.46785),
}
city_names = list(locations.keys())
coordinates = list(locations.values())
n = len(city_names)
# 2. Calcular la matriz de distancias en kilómetros
```

```
distance_matrix = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        if i != j:
            distance_matrix[i][j] = geodesic(coordinates[i],
coordinates[j]).kilometers
# 3. Función para calcular el MST usando el algoritmo de Prim
def compute_mst_weight(distance_matrix, vertices):
    if not vertices:
        return 0.0
    in_tree = {vertices[0]}
    not_in_tree = set(vertices[1:])
    mst_weight = 0.0
    while not_in_tree:
        min_edge = float('inf')
        for u in in_tree:
            for v in not_in_tree:
                if distance_matrix[u][v] < min_edge:
                    min_edge = distance_matrix[u][v]
                    chosen_edge = (u, v)
        mst_weight += min_edge
        in_tree.add(chosen_edge[1])
        not_in_tree.remove(chosen_edge[1])
    return mst_weight
#  Método del 1-árbol
# Paso 1: Escoger "MORÓN" como el vértice a eliminar
v = "MORÓN"
v_index = city_names.index(v)
# Paso 2: Seleccionar las dos aristas de menor coste incidentes a "MORÓN"
edges_from_v = [(i, distance_matrix[v_index][i]) for i in range(n) if i != v_index]
edges_from_v.sort(key=lambda x: x[1])
edge1, edge2 = edges_from_v[0], edges_from_v[1]
sum_edges = edge1[1] + edge2[1]
print("Aristas seleccionadas desde", v, ":")
print(f"{v} -> {city_names[edge1[0]]} : {edge1[1]:.2f} km")
print(f"{v} -> {city_names[edge2[0]]} : {edge2[1]:.2f} km")
# Paso 3: Calcular el MST en el subgrafo inducido por los vértices restantes (sin
MORÓN)
remaining = [i for i in range(n) if i != v_index]
mst_weight = compute_mst_weight(distance_matrix, remaining)
print("\nPeso del MST (sin", v, "): {:.2f} km".format(mst_weight))
# Paso 4: La cota inferior (1-árbol) es la suma del peso del MST y de las dos aristas
de MORÓN
lower_bound_km = mst_weight + sum_edges
lower_bound_nm = lower_bound_km / 1.852  # Conversión: 1 milla náutica = 1.852 km
print("\nCota inferior del TSP (método 1-árbol):")
print("{:.2f} km".format(lower_bound_km))
print("{:.2f} millas náuticas".format(lower_bound_nm))
```

El programa nos entrega los siguientes resultados:

```
Aristas seleccionadas desde MORÓN:
MORÓN -> ROTA: 86.74 km
MORÓN -> MÁLAGA: 155.83 km

Peso del MST (sin MORÓN): 877.79 km

Cota inferior del TSP (método 1-árbol):
1120.35 km
604.94 millas náuticas
```

Código para el método de Christófides:

```python
import networkx as nx
import matplotlib.pyplot as plt
from geopy.distance import geodesic
#  Definición de las ciudades y sus coordenadas
locations = {
    "MORÓN": (37.3597, -5.96755),
    "ROTA": (36.639475, -6.346078),
    "CEUTA": (35.89902, -5.28055),
    "MÁLAGA": (36.7131, -4.4128),
    "PEÑÓN DE VÉLEZ": (35.17785, -4.29972),
    "ISLA DE ALBORÁN": (35.937725, -3.0365),
    "MELILLA": (35.29444, -2.9336),
    "CHAFARINAS": (35.1809, -2.433905),
    "CARTAGENA": (37.595451, -0.98128),
    "ALMERÍA": (36.83433, -2.46785),
}
# Creación del grafo completo
G = nx.Graph()
# Agregar nodos con sus posiciones (usando (longitud, latitud))
for city in locations:
    G.add_node(city, pos=(locations[city][1], locations[city][0]))
# Agregar todas las aristas con su distancia geodésica como peso
for city1 in locations:
    for city2 in locations:
        if city1 != city2:
            dist = geodesic(locations[city1], locations[city2]).kilometers
            G.add_edge(city1, city2, weight=dist)
# Paso 1: Calcular el MST
mst = nx.minimum_spanning_tree(G, weight="weight")
# Paso 2: Identificar nodos de grado impar en el MST
odd_nodes = [v for v in mst.nodes() if mst.degree(v) % 2 == 1]
print("Nodos con grado impar:", odd_nodes)
# Paso 3: Construir un subgrafo completo con los nodos impares
odd_graph = nx.Graph()
for i, u in enumerate(odd_nodes):
    for j, v in enumerate(odd_nodes):
        if i < j:
            odd_graph.add_edge(u, v, weight=G[u][v]['weight'])
# Paso 4: Calcular el emparejamiento perfecto de peso mínimo
matching = nx.algorithms.matching.min_weight_matching(odd_graph, weight='weight')
```

```
print("Emparejamiento perfecto mínimo:", matching)
# Paso 5: Combinar el MST con el emparejamiento para formar un grafo Euleriano
# Se crea un MultiGrafo ya que algunas aristas pueden repetirse.
multigraph = nx.MultiGraph()
multigraph.add_edges_from(mst.edges(data=True))
# Agregar las aristas del emparejamiento
for u, v in matching:
    multigraph.add_edge(u, v, weight=G[u][v]['weight'])
# Paso 6: Encontrar un circuito Euleriano en el grafo resultante
# Se elige un nodo arbitrario como fuente
source = list(mst.nodes())[0]
eulerian_circuit = list(nx.eulerian_circuit(multigraph, source=source))
print("Circuito Euleriano:", eulerian_circuit)
#Paso 7: obtener un ciclo Hamiltoniano Se recorren los nodos del circuito euleriano,
agregando solo los no visitados
tsp_path_cleaned = []
visited = set()
for u, v in eulerian_circuit:
    if u not in visited:
        tsp_path_cleaned.append(u)
        visited.add(u)
# Cerrar el ciclo regresando a Morón
tsp_path_cleaned.append(tsp_path_cleaned[0])
print("Ruta TSP:", tsp_path_cleaned)
# Cálculo de la distancia total de la ruta TSP aproximada
total_distance = sum(G[u][v]['weight'] for u, v in zip(tsp_path_cleaned,
tsp_path_cleaned[1:]))
total_distance_nautical = total_distance / 1.852  # Conversión a millas náuticas
print(f"Distancia total en millas náuticas: {total_distance_nautical:.2f} nm")
```

El programa al ejecutar este código aporta este resultado:

```
Nodos con grado impar: ['MORÓN', 'CEUTA', 'MÁLAGA', 'MELILLA', 'CHAFARINAS',
'CARTAGENA']
Emparejamiento perfecto mínimo: {('CEUTA', 'MORÓN'), ('CHAFARINAS', 'MELILLA'),
('MÁLAGA', 'CARTAGENA')}
Circuito Euleriano: [('MORÓN', 'CEUTA'), ('CEUTA', 'PEÑÓN DE VÉLEZ'), ('PEÑÓN DE
VÉLEZ', 'MELILLA'), ('MELILLA', 'CHAFARINAS'), ('CHAFARINAS', 'MELILLA'),
('MELILLA', 'ISLA DE ALBORÁN'), ('ISLA DE ALBORÁN', 'ALMERÍA'), ('ALMERÍA',
'CARTAGENA'), ('CARTAGENA', 'MÁLAGA'), ('MÁLAGA', 'CEUTA'), ('CEUTA', 'ROTA'),
('ROTA', 'MORÓN')]
Ruta TSP: ['MORÓN', 'CEUTA', 'PEÑÓN DE VÉLEZ', 'MELILLA', 'CHAFARINAS', 'ISLA DE
ALBORÁN', 'ALMERÍA', 'CARTAGENA', 'MÁLAGA', 'ROTA', 'MORÓN']
Distancia total en millas náuticas: 763.36 nm
```

Para la implementación del método de inserción más cercana y que grafique cada uno de los pasos para las figuras expuestas en el apartado 4 de este trabajo, se ha empleado el siguiente código:

```
# Recargar librerías
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from geopy.distance import geodesic
import networkx as nx
# Definición de las ciudades y sus coordenadas
locations = {
    "MORÓN": (37.3597, -5.96755),
    "ROTA": (36.639475, -6.346078),
    "CEUTA": (35.89902, -5.28055),
    "MÁLAGA": (36.7131, -4.4128),
    "PEÑÓN DE VÉLEZ": (35.17785, -4.29972),
    "ISLA DE ALBORÁN": (35.937725, -3.0365),
    "MELILLA": (35.29444, -2.9336),
    "CHAFARINAS": (35.1809, -2.433905),
    "CARTAGENA": (37.595451, -0.98128),
    "ALMERÍA": (36.83433, -2.46785),
}
city_names = list(locations.keys())
n = len(city_names)
coordinates = list(locations.values())
# Calcular la matriz de distancias en kilómetros usando geodesic
distance_matrix = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        if i != j:
            distance_matrix[i][j] = geodesic(coordinates[i],
coordinates[j]).kilometers
# Crear un diccionario de posiciones para graficar
pos = {city: (locations[city][1], locations[city][0]) for city in locations}
# Creamos un grafo completo para utilizarlo como fondo en las gráficas
G = nx.Graph()
for city in locations:
    G.add_node(city, pos=pos[city])
def plot_cycle(cycle, step):
    cycle_cities = [city_names[i] for i in cycle]
    # Construir las aristas del ciclo (conecta cada par consecutivo)
    edges = [(cycle_cities[i], cycle_cities[i+1]) for i in range(len(cycle_cities)-
1)]
    plt.figure(figsize=(10,6))
    # Dibujar nodos y etiquetas usando las posiciones definidas en 'pos'
    nx.draw_networkx_nodes(G, pos, node_size=700, node_color='lightblue')
    nx.draw_networkx_labels(G, pos, font_size=10, font_weight='bold')
    # Dibujar las aristas del ciclo actual en rojo
    nx.draw_networkx_edges(G, pos, edgelist=edges, edge_color='red', width=2)
    plt.title("Paso {}: Ciclo actual".format(step))
```

```
    plt.xlabel("Longitud (Oeste)")
    plt.ylabel("Latitud (Norte)")
    plt.axis('off')
    plt.show()
# Función del método de inserción más cercana y que grafica en cada paso
def insercion_mas_cercana_con_plots(distance_matrix, start_city):
    start_index = city_names.index(start_city)
    remaining = set(range(n))
    remaining.remove(start_index)
    # Paso 1: A partir de MORÓN, hallar la ciudad más cercana
    nearest = min(remaining, key=lambda i: distance_matrix[start_index][i])
    remaining.remove(nearest)
    # Paso 2: Formar el ciclo inicial: [MORÓN, ciudad_cercana, MORÓN]
    cycle = [start_index, nearest, start_index]
    plot_cycle(cycle, step=0)  # Graficamos el ciclo inicial
    step = 1
    # Paso 3: Mientras queden ciudades por insertar, elegir la inserción que minimice
la conexión
    while remaining:
        best_v = None  # vértice candidato fuera del ciclo
        best_u = None  # vértice del ciclo que da la conexión mínima
        best_dist = float('inf')
        # Buscar, para cada vértice no insertado, la mínima conexión con algún
vértice del ciclo
        for v in remaining:
            for i in range(len(cycle) - 1):
                u = cycle[i]
                d = distance_matrix[u][v]
                if d < best_dist:
                    best_dist = d
                    best_v = v
                    best_u = u
        # Insertar best_v en el ciclo inmediatamente DESPUÉS del vértice best_u
        pos_index = cycle.index(best_u)
        cycle.insert(pos_index + 1, best_v)
        remaining.remove(best_v)
        # Graficar el ciclo actualizado
        plot_cycle(cycle, step=step)
        step += 1
    return cycle
# Ejecutar el algoritmo de inserción más cercana tomando MORÓN como origen
final_cycle_indices = insercion_mas_cercana_con_plots(distance_matrix,
start_city="MORÓN")
final_cycle_cities = [city_names[i] for i in final_cycle_indices]
# Calcular la distancia total del ciclo en kilómetros
total_distance_km =
sum(distance_matrix[final_cycle_indices[i]][final_cycle_indices[i+1]] for i in
range(len(final_cycle_indices)-1))
# Convertir la distancia total a millas náuticas
total_distance_nautical = total_distance_km / 1.852
```

```
print("Ciclo final TSP (cota superior, inserción más cercana, origen MORÓN):")
print(final_cycle_cities)
print("\nDistancia total del ciclo:")
print("{:.2f} millas náuticas".format(total_distance_nautical))
```

El programa al ejecutarse; a parte de las figuras empleadas en el punto 4.3.1, aporta lo siguiente:

```
Ciclo final TSP (cota superior, inserción más cercana, origen MORÓN):
['MORÓN', 'ROTA', 'CEUTA', 'PEÑÓN DE VÉLEZ', 'MELILLA', 'ISLA DE ALBORÁN',
'ALMERÍA', 'CARTAGENA', 'CHAFARINAS', 'MÁLAGA', 'MORÓN']

Distancia total del ciclo:
809.03 millas náuticas
```

Para la implementación del método del árbol se ha utilizado el siguiente código:

```
#Abrir librerías
import networkx as nx
import matplotlib.pyplot as plt
from geopy.distance import geodesic
# Definición de las ciudades y sus coordenadas
locations = {
    "MORÓN": (37.3597, -5.96755),
    "ROTA": (36.639475, -6.346078),
    "CEUTA": (35.89902, -5.28055),
    "MÁLAGA": (36.7131, -4.4128),
    "PEÑÓN DE VÉLEZ": (35.17785, -4.29972),
    "ISLA DE ALBORÁN": (35.937725, -3.0365),
    "MELILLA": (35.29444, -2.9336),
    "CHAFARINAS": (35.1809, -2.433905),
    "CARTAGENA": (37.595451, -0.98128),
    "ALMERÍA": (36.83433, -2.46785),
}
# Crear el grafo completo G con pesos dados por la distancia geodésica (en km)
G = nx.Graph()
# Agregar nodos; para la visualización usamos (longitud, latitud)
for city, coord in locations.items():
    G.add_node(city, pos=(coord[1], coord[0]))
# Agregar aristas con su peso
for city1 in locations:
    for city2 in locations:
        if city1 != city2:
            d = geodesic(locations[city1], locations[city2]).kilometers
            G.add_edge(city1, city2, weight=d)
# Paso 1: Calcular el MST del grafo completo
mst = nx.minimum_spanning_tree(G, weight='weight')
# Paso 2: Duplicar las aristas del MST para obtener un multigrafo Euleriano
multigraph = nx.MultiGraph()
multigraph.add_nodes_from(mst.nodes(data=True))
for u, v, data in mst.edges(data=True):
 # Se agregan dos copias de cada arista
    multigraph.add_edge(u, v, weight=data['weight'])
    multigraph.add_edge(u, v, weight=data['weight'])
#  Paso 3: Encontrar un circuito Euleriano en el multigrafo
```

```
# Se toma como nodo fuente Morón
eulerian_circuit = list(nx.eulerian_circuit(multigraph, source=list(mst.nodes())[0]))
# Paso 4: Obtener un ciclo hamiltoniano (solución TSP aproximada)
hamiltonian_cycle = []
visited = set()
for u, v in eulerian_circuit:
    if u not in visited:
        hamiltonian_cycle.append(u)
        visited.add(u)
# Se añade el nodo inicial para cerrar el ciclo
hamiltonian_cycle.append(hamiltonian_cycle[0])
# --- Paso 5: Calcular el coste total del ciclo ---
total_cost = 0
for i in range(len(hamiltonian_cycle)-1):
    u = hamiltonian_cycle[i]
    v = hamiltonian_cycle[i+1]
    total_cost += G[u][v]['weight']
total_cost_km = total_cost
total_cost_nm = total_cost_km / 1.852  # Conversión a millas náuticas
print("Ciclo hamiltoniano aproximado (Método del árbol):")
print(hamiltonian_cycle)
print("Costo total: {:.2f} millas náuticas".format(total_cost_nm))
```

El resultado en este caso aportado por el programa es el siguiente:

```
Ciclo hamiltoniano aproximado (Método del árbol):
['MORÓN', 'ROTA', 'CEUTA', 'PEÑÓN DE VÉLEZ', 'MELILLA', 'CHAFARINAS', 'ISLA DE
ALBORÁN', 'ALMERÍA', 'CARTAGENA', 'MÁLAGA', 'MORÓN']
Costo total: 728.61 millas náuticas
```