Universidad de Valladolid

Facultad de Ciencias

Trabajo Fin de Grado

Grado en Matemáticas

Reconstrucción de trayectorias de aeronaves usando heurísticas de mejora para resolver una versión del *problema del viajante* (TSP)

Autor: Juan Manuel Velasco Heras

Tutor: Pedro César Álvarez Esteban

Agradecimientos

A Boeing Research and Technology Europe y en especial al profesor D. Pedro César Álvarez Esteban por haber aceptado la dirección de este trabajo.

Resumen

El tráfico aéreo mundial es cada vez más concurrido. Solo en Europa, Eurocontrol, en su último informe de previsiones [Eurocontrol, 2017], pronosticaba que se producirían 11,6 millones de desplazamientos aéreos durante 2017, aumentando en un 14 % para 2023. Es por ello necesario llevar a cabo una profunda renovación de los sistemas de gestión del tráfico aéreo, apostando por nuevas tecnologías *bigdata* capaces de gestionar las ingentes cantidades de datos generados. El proyecto AIRPORTS (CIEN, 2015), liderado por *Boeing Research & Technology Europe*, involucra a la Universidad de Valladolid en esta tarea, así como en la reconstrucción de trayectorias de aeronaves a partir de los datos de seguimiento obtenidos mediante la nueva tecnología ADS-B (Automatic Dependent Surveillance - Broadcast), que, a pesar de suponer un salto de calidad con respecto a las antiguas comunicaciones vía radar, presenta algunos problemas de alineamiento temporal que afectan en ocasiones a la veracidad de esas reconstrucciones.

En este sentido, el presente documento tiene como objetivo principal estudiar esta problemática, así como proponer una solución basada en el modelado de las trayectorias de aeronaves a modo de una variante del denominado *problema del viajante* (TSP) en la que el nodo inicial y el final no coinciden. Para poder ser expuesta, es necesario el previo estudio de los fundamentos teóricos que componen este problema, así como del amplio conjunto de métodos y estrategias orientadas a su resolución. De entre todas ellas, este trabajo se centra en las denominadas *heurísticas de mejora local* e incluye sendos estudios comparativos que analizan la eficacia de estos algoritmos en diferentes situaciones con el fin de concluir su nivel de adecuación al supuesto práctico arriba descrito.

Palabras clave: TSP, ADS-B, algoritmo, reconstrucción, aeronaves, ventanas.

Índice general

Introducción

Desde principios de siglo, el tráfico aéreo mundial viene siendo objeto de un crecimiento constante en su densidad. La *gestión del tráfico aéreo*, del inglés *Air Traffic Management* (ATM), pasa a cobrar cada vez más importancia en un entorno en el que la actividad, con el paso del tiempo, es cada vez mayor. A esta dinámica creciente hay que sumarle la mayor complejidad de sus actores, de diversa tipología (aeropuertos, aerolíneas, aeronaves, etc) [Alonso-Isla et al., 2018], resultando en una demanda en aumento de tecnologías más avanzadas que se adapten a las cada vez mayores exigencias del sector.

El organismo encargado de la gestión del tráfico aéreo en Europa es Eurocontrol, creado en 1963 por la *International Civil Aviation Organization* (ICAO) para constituir una única corporación con responsabilidades sobre todo el espacio aéreo europeo [Grushka-Cockayne y De Reyck, 2009]. En sus inicios, los distintos gobiernos nacionales se negaron a renunciar a parte de su soberanía en favor de una gestión comunitaria del tráfico aéreo, derivando en un espacio aéreo dividido en bloques o sectores nacionales.

La existencia de estas fronteras estatales, unida a la necesidad de modernizar los sistemas ATM continentales, derivan en pérdidas estimadas entre 200 y 300 millones de euros anuales con respecto a otras estructuras homólogas [Alonso-Isla et al., 2018]. Por ello, surge en marzo de 2004 la iniciativa *Single European Sky* (SES), de la mano de la Comisión Europea [Grushka-Cockayne y De Reyck, 2009], para abordar una reestructuración del espacio aéreo europeo que establezca una nueva división por bloques funcionales (acordes a los flujos del tráfico), reemplazando al anterior sistema por bloques nacionales, mucho más ineficiente; así como para modernizar los sistemas de control del tráfico aéreo o *Air Traffic Control* (ATC).

En este afán renovador, se incluye una de las tecnologías de seguimiento del tráfico aéreo más recientes: las *Automatic Dependent Surveillance - Broadcast* (ADS-B), que suponen un importante paso hacia delante en esta materia. Su uso pasó a ser obligatorio para algunas de las aeronaves que transitan por Europa a partir de 2017, a fin de ser adaptadas para su correcta implementación y funcionamiento. La tecnología ADS-B se caracteriza por utilizar los sistemas de navegación vía satélite o *Global Navigation Satellite System* (GNSS) como método por el que las aeronaves obtienen su posición para posteriormente comunicarla a través del aire en forma de paquetes de información que reciben el nombre de *señales o mensajes ADS-B*. Estas comunicaciones son recibidas por diversas estaciones receptoras en tierra, así como por otras aeronaves en vuelo cercanas.

Cuando una señal es recibida por un receptor, éste le asocia una marca temporal o

timestamp a fin de poder establecer una ordenación de los mensajes ADS-B asociados a un mismo vuelo o trayectoria. El almacenamiento ordenado de estos datos es de gran utilidad para hacer reconstrucciones posteriores de las trayectorias voladas. Una de las múltiples líneas de investigación del proyecto AIRPORTS (CIEN, 2015), liderado por *Boeing Research & Technology Europe* trata las diversas formas fusionar los datos de varias fuentes (receptores) de manera que las reconstrucciones obtenidas se hagan de la forma más eficiente posible.

A pesar de las ventajas de ADS-B, entre las que se encuentran la veracidad de sus datos de geolocalización (al obtener la aeronave su posición a través de una red de satélites de comunicaciones), o su alta frecuencia de comunicaciones (de hasta 2 mensajes ADS-B por segundo), esta tecnología cuenta, por un lado, con la potencial aparición de problemas de escalabilidad en los sistemas ATM, obligando a la utilización de tecnologías *bigdata* capaces de hacer frente a las altas cargas de procesamiento diarias producidas por la enorme cantidad de datos a tratar.

Por otro lado, y es en lo que se enfoca la presente memoria, la tecnología ADS-B cuenta con problemas de alineamiento temporal en los datos producidos por la no sincronización de los receptores de distintas organizaciones que nutren de datos a los sistemas ATM, y por los retardos que sufren las señales viajando a través del aire hasta alcanzar un receptor. Como consecuencia, el orden en que son recibidas estas señales, y por tanto, asociadas a una marca temporal, no se corresponde con la verdadera disposición en que fueron enviados los mensajes.

En este sentido, el primer objetivo de este Trabajo Fin de Grado es proponer un modelo que permita detectar, así como corregir, los posibles problemas que se den en el procesado de conjuntos de mensajes ADS-B procedentes de diferentes vuelos.

El modelo de solución que se propone pasa por transformar esta problemática en una variante del conocido *problema del viajante* en la que el nodo de partida y el de llegada no coinciden, y considerando como nodos cada uno de los mensajes ADS-B asociados a la trayectoria. Esta transformación requiere un análisis detallado del problema anterior, estudiando sus características computacionales así como las diferentes técnicas y métodos que lo resuelven. Dada la gran variedad de algoritmos existentes, es también objetivo de este trabajo profundizar sobre el funcionamiento y características de un tipo muy conocido de estos métodos: las *heurísticas de mejora local*.

A lo largo de los tres primeros capítulos se desarrollan los aspectos fundamentales a tratar del problema del viajante, con el fin de servir a modo de base para construir, en adelante, la propuesta de solución que esta memoria plantea.

Capítulo 1

El problema del viajante: introducción y estudio de su complejidad

El presente capítulo es el primero de una serie de tres que tratan en profundidad el conocido *problema del viajante* o *Traveling Salesman Problem* (TSP). Su análisis viene motivado por su aplicación al modelado del problema de la reordenación de los mensajes ADS-B de una trayectoria, anunciado en la introducción, y del que se hablará con mayor profundidad en el Capítulo 4.

A lo largo de este primer capítulo de la memoria se recogen los fundamentos teóricos del TSP. Primeramente, se introduce formalmente el problema así como los conceptos básicos sobre los que se apoya (ver Sección 1.1). A continuación, se exponen sus rasgos históricos más destacados, guiándonos para ello del enfoque dado en Lawler et al. [1992] (ver Sección 1.2). La Sección 1.3, se centra en el análisis de la dificultad de resolución del TSP, para lo cual se exponen algunos resultados básicos de la teoría de complejidad computacional. Por último, se presentan algunas aplicaciones prácticas del TSP en la Sección 1.4.

1.1. Definición del problema

Para comenzar a estudiar el problema del viajante es necesario introducir algunos conceptos de la teoría de grafos [Berge, 1976]. Primero, se presenta el concepto de grafo.

Definición 1.1. *Se define un* ***grafo*** $G = (V, E)$ *como un conjunto* V *de puntos denominados* ***vértices*** *o* ***nodos*** *y un conjunto* $E \subset V \times V$ *de conexiones que unen pares de vértices del grafo, denominadas* ***aristas****. Se dice que el grafo es* ***no dirigido*** *si para cada* $(v_i, v_j) \in E$ *se tiene que* $(v_j, v_i) \in E$*. En caso contrario, se dice que el grafo es* ***dirigido****. Un grafo se denomina* ***completo*** *si* $\{(v_i, v_j) \in V \times V \mid i \neq j\} \subset E$*.*

A continuación, se introduce el término *camino sobre un grafo*.

Definición 1.2. *Sea $G = (V, E)$ un grafo. Se dice que C es un* ***camino*** *sobre G si, $C = (v_1, v_2, \ldots, v_r) \in V^r \, (= V \times \overset{r}{\cdots} \times V)$, donde $r \in \mathbb{N}$, $r \geq 2$. Si además C verifica que $v_r = v_1$ se dice que C es un* ***camino cerrado*** *o* ***circuito*** *sobre G. En caso contrario, se dice que C es un* ***camino abierto****.*

El problema del viajante consiste en la búsqueda de un camino con una serie de propiedades. Una de ellas es que atraviese una sola vez todos y cada uno de los vértices del grafo. La Definición 1.3 abarca este tipo de caminos cuando son abiertos, mientras que la Definición 1.4 lo hace cuando son cerrados.

Definición 1.3. *Sea $G = (V, E)$ un grafo y sea $C = (v_1, v_2, \ldots, v_r)$ un camino sobre G. Se dice que C es un* ***camino hamiltoniano*** *sobre G si $v_i \neq v_j$ para cada i, j con $1 \leq i < j \leq r$ y $r = |V|$.*

Definición 1.4. *Sea $G = (V, E)$ un grafo y sea $C = (v_1, v_2, \ldots, v_{r-1}, v_1)$ un circuito sobre G. Se dice que C es un* ***ciclo o circuito hamiltoniano*** *sobre G si $v_i \neq v_j$ para cada i, j con $1 \leq i < j \leq r - 1$ y $r - 1 = |V|$.*

Algunos grafos, por construcción, no pueden albergar ciclos hamiltonianos en su interior. Es por ello, que los que sí lo hacen reciben el nombre de *grafos hamiltonianos*.

A continuación, se presenta el concepto de permutación.

Definición 1.5. *Una* ***permutación*** *π sobre un conjunto $N = \{1, 2, \ldots, n\}$ con $n \in \mathbb{N}$ es una función biyectiva que va de N en si mismo. Se dice además que π es* ***circular*** *si para todo $E \subsetneq N$ no vacío, existe $e \in E$ de manera que $\pi(e) \notin E$.*

Nótese que, todo ciclo hamiltoniano puede representarse como una permutación circular π sobre el $N = \{1, 2, \ldots, n\}$, siendo n el número de vértices del grafo y donde cada $k \in N$ se asocia a un nodo. Así, $\pi(i)$ será el vértice que se visita inmediatamente tras pasar el ciclo hamiltoniano por el nodo i.

Presentada la terminología básica, se expone en la Definición 1.6 el *problema del viajante*, objeto de estudio de este capítulo.

Definición 1.6. *Sea $G = (V, E)$ un grafo completo donde $V = \{v_1, v_2, \ldots, v_n\}$ y $E = \{(v_i, v_j) \,|\, v_i, v_j \in V, \, i \neq j\}$, sea $C = (c_{ij})$ la matriz $n \times n$ tal que $c_{ij} \geq 0$ es la longitud o coste asociado a la arista (v_i, v_j), con $i \neq j$, y $c_{ii} = 0$ si $1 \leq i \leq n$, y sea Π el conjunto de todas las permutaciones circulares sobre el conjunto $N = \{1, 2, \ldots, n\}$. El* ***problema del viajante (TSP)*** *consiste en encontrar una permutación circular $\pi^* \in \Pi$ sobre N tal que*

$$\sum_{i=1}^{n} c_{i\pi^*(i)} = \min_{\pi \in \Pi} \sum_{i=1}^{n} c_{i\pi(i)}. \tag{1.1.1}$$

Por tanto, el objetivo es encontrar el ciclo hamiltoniano (también denominado *ruta* en el contexto del TSP) de mínima longitud, esto es, que minimice (1.1.1).

El problema del viajante pertenece a la familia de los problemas de optimización combinatoria (o discreta), por lo que no es posible resolverlo a través del cálculo diferencial, a diferencia de otras optimizaciones sobre espacios continuos donde para maximizar o minimizar la función objetivo basta con encontrar sus puntos críticos.

A lo largo de la memoria aparece repetidas veces el término *instancia*, entendido como supuesto o especificación concreta del modelo formal que plantea un problema. Se dice así que el problema es la abstracción que se concreta a través de sus instancias. En el caso del TSP, éstas quedan caracterizadas en la Definición 1.7.

Definición 1.7. *Una **instancia $\mathcal{I}$ del problema TSP** se define como un par (G, C) donde $G = (V, E)$ es un grafo completo con $V = \{v_1, v_2, \ldots, v_n\}$ como conjunto de vértices y $E = \{(v_i, v_j) \,|\, v_i, v_j \in V,\, i \neq j\}$ como conjunto de aristas; y donde $C = (c_{ij})_{1 \leq i,j \leq n}$ es una matriz verificando que $c_{ij} \geq 0$ es la longitud o coste asociado a la arista $(v_i, v_j), i \neq j$, y $c_{ii} = 0$ si $1 \leq i \leq n$.*

*Se define en estas condiciones el **problema TSP simétrico (STSP)** como aquel cuyas instancias $\mathcal{I} = (G, C)$ se corresponden con las del problema TSP tales que $c_{ij} = c_{ji}$ para cada $1 \leq i, j \leq n$. Por el contrario, las instancias del problema TSP que no verifican esta condición pertenecen al denominado **problema TSP asimétrico (ATSP)**.*

Nótese que en el caso del problema STSP, la matriz C es simétrica para todas las instancias del mismo, mientras que no lo es para las instancias del problema ATSP.

De ahora en adelante, los desarrollos expuestos a lo largo de este trabajo se centrarán en el problema TSP simétrico, por su adecuada adaptación al supuesto práctico que esta memoria estudia en el Capítulo 4. Esta simetría se traduce en que la longitud de la arista que une un punto con otro en un sentido es la misma que la longitud de la que lo hace en sentido contrario.

Además, dentro del problema STSP haremos hincapié en un tipo especial de instancias en el que el coste de las aristas verifica, además, las propiedades de una distancia considerando los nodos (también denominados *ciudades* en el contexto del TSP) como puntos del plano. Estas instancias son a su vez instancias del denominado problema TSP métrico.

Definición 1.8. *Sea $G = (V, E)$ un grafo completo con $|V| = n$ y $C = (c_{ij})$ una matriz simétrica de tamaño $n \times n$ donde $c_{ij} \geq 0$ es el coste de la arista que une v_i con v_j, siendo $v_i, v_j \in V$. Se dice que $\mathcal{I} = (G, C)$ es una instancia del llamado **TSP métrico** si se verifica la desigualdad triangular, esto es,*

$$c_{ij} + c_{jk} \geq c_{ik} \quad \forall i, j, k,\, 1 \leq i, j, k \leq n.$$

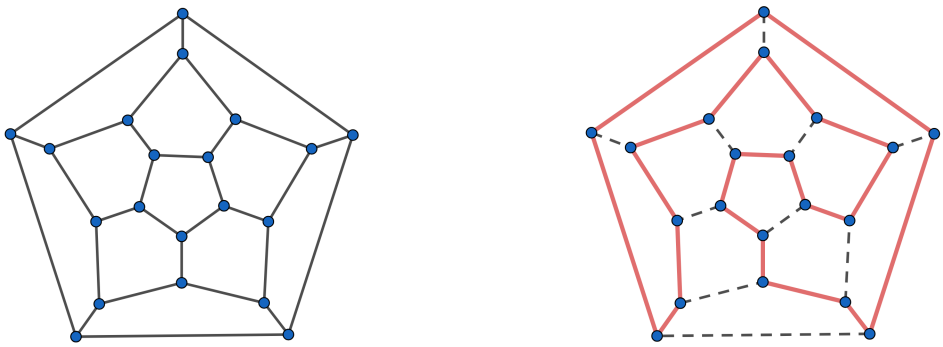
Quizás el caso particular más importante del TSP métrico, y el que se tratará en este trabajo de forma más profunda por su mejor adaptación a nuestro supuesto, es el denominado **TSP euclídeo**, que considera los nodos del grafo como puntos en el espacio $\mathbb{R}^n$ (aunque en nuestro caso nos centraremos en el plano $\mathbb{R}^2$), y donde el coste de las aristas viene dado por la distancia euclídea entre los nodos que unen.

1.2. Historia del TSP

En esta sección se resumen, en líneas generales, los aspectos más importantes que caracterizan al TSP, siguiendo el enfoque de Lawler et al. [1992].

El problema del viajante, cuya traducción literal del inglés es *problema del vendedor ambulante*, recibe popularmente su nombre de la necesidad que se le presenta a estos profesionales de visitar un determinado conjunto de ciudades por cuestiones de trabajo para después retornar a su lugar de partida, con un claro interés por hacer este trayecto invirtiendo el menor tiempo posible. Suponiendo longitud y tiempo de viaje proporcionales, esta aspiración se traduce en encontrar el camino de mínima distancia que atraviese todas las ubicaciones prefijadas. Esta idea no es más que el objetivo final del TSP.

Los primeros antecedentes del TSP parten del desarrollo de la teoría de grafos a lo largo del siglo XIX, de la mano de matemáticos como Kirkman, quién en 1856 fue el primero en considerar los ciclos hamiltonianos en un contexto general, estableciendo condiciones para su existencia sobre grafos poliédricos. Ese mismo año, Sir William Rowan Hamilton descubrió un álgebra no conmutativa, que denominó *The Icosian Calculus*, cuyos elementos son las acciones o posibles movimientos sobre los vértices del dodecaedro regular, y sobre el cual plantea el problema de encontrar ciclos hamiltonianos bajo ciertas restricciones, como por ejemplo, estableciendo su paso obligado por cinco vértices determinados de forma consecutiva. A raíz de su teoría, Hamilton confeccionó un juego de inteligencia que denominó *The Icosian Game* en el que desafiaba al jugador a encontrar un camino con determinadas propiedades sobre el grafo del dodecaedro regular (ver Figura 1.1).



(a) Grafo del dodecaedro regular

(b) Ciclo hamiltoniano sobre el grafo del dodecaedro regular

Figura 1.1: El juego de inteligencia *The Icosian Game* reta al jugador a encontrar ciclos hamiltonianos sobre el grafo del dodecaedro regular bajo ciertas restricciones

En 1930, Menger dio mayor importancia al coste del camino por ser la medida de la longitud de curvas en el espacio uno de sus temas de estudio favoritos. Surge entonces el denominado *problema del mensajero* o *messenger problem*, que consiste en encontrar el camino abierto más corto que una todos los vértices de un grafo (pensemos en un mensajero que debe entregar paquetes en varias direcciones prefijadas en el menor tiempo

posible). Se diferencia sutilmente del TSP en que no se requiere que las ciudades de partida y de llegada coincidan.

En 1954, George Dantzig, Ray Fulkerson y Selmer Johnson, miembros del grupo de matemáticos de la *RAND Corporation*, recogen el testimonio de Both Flood, quien atestigua que en 1937 Hassler Whitney, en una de sus conferencias, introdujo el nombre de *Traveling Salesman Problem* para denominar al problema igualmente nombrado. Dantzig, Fulkerson y Johnson presentan además uno de los primeros algoritmos de resolución del problema: el algoritmo de *planos cortantes*, basado en programación lineal entera.

Durante esa década el TSP se popularizó en América surgiendo nuevas líneas de investigación. La *RAND Corporation* se convirtió en el principal núcleo investigador y anunció el pago de importantes cantidades económicas por la resolución de instancias concretas del problema a fin de fomentar el desarrollo de nuevos métodos.

Posteriormente, tres matemáticos: Cook en 1971, Karp en 1972 y Lewis en 1973; desarrollaron sendos trabajos en los que indagaban sobre la dificultad de resolución de una serie de problemas, catalogados como $\mathcal{NP}$*-hard* o *non-deterministic polynomial time problems* (ver Subsección 1.3.1), caracterizados por ser su resolución computacionalmente costosa, no estando probada la existencia de un algoritmo con complejidad computacional polinomial que garantice encontrar una solución (óptima) del problema. De sus estudios se comienza a entrever la equivalencia de todos estos problemas, pues un algoritmo polinomial que resolviese alguno de ellos también resolvería el resto. Además, fue posible probar que ciertos casos concretos del TSP eran $\mathcal{NP}$*-hard*, entre ellos el problema TSP euclídeo, mencionado en la sección anterior.

La aparente dificultad del TSP motiva la aparición de dos corrientes de algoritmos. La primera persigue alcanzar la solución óptima (una ruta de longitud mínima) aun a riesgo de no encontrarla en un tiempo razonable. La segunda, sin embargo, prioriza el coste computacional a la calidad de la solución, conformándose con soluciones que simplemente se aproximen a la más corta, sin llegar a ser óptimas. En la primera opción se encuentran los algoritmos de *ramificación y acotación* (o *branch and bound*) , así como la programación dinámica, que solo sirvió para resolver pequeñas instancias del problema. Para la segunda opción, surgen los métodos basados en *heurísticas de mejora* (cuyo análisis es objeto de este Trabajo Fin de Grado), que dada una primera solución inicial realizan modificaciones sobre ella con el fin de obtener nuevas soluciones más cortas.

En 1980, el problema TSP con 318 ciudades fue resuelto por Crowder y Padberg mediante la división del problema en 3 fases. La primera consistió en el uso del *algoritmo de Lin-Kernighan* (que se introduce en la Sección 2.2.3) para encontrar una ruta de partida y se ejecuta un procedimiento basado en *símplex* sobre un problema de programación lineal con restricciones obteniendo una cota inferior de la longitud de la solución óptima. En la segunda fase se fija un subconjunto de variables para, en la tercera fase, hacer, de nuevo, uso de programación lineal. Se sabe que esta tercera fase conllevó aproximadamente 6 minutos de tiempo de CPU.

Más recientemente, destacan los supuestos que el grupo de investigación formado por David Applegate, Robert Bixby, Vasek Chvátal y William Cook logró resolver a través de

un algoritmo de creación propia que denominaron *Concorde*. A través de este método han obtenido la solución de instancias de tamaño no resuelto hasta entonces. La más reciente es una con 85.900 ciudades en el año 2006 [Cook, 2012].

1.3. Análisis de la complejidad computacional del problema del viajante

Uno de los aspectos por los que más es conocido el problema del viajante es por la alta dificultad que entraña su resolución. Esta sección tiene como objetivo profundizar sobre la teoría de la complejidad computacional para así estudiar esta ya mencionada característica del TSP.

1.3.1. Introducción a la complejidad computacional

La teoría de la complejidad computacional estudia la dificultad que entraña la resolución de problemas computables mediante algoritmos (funciones o subrutinas informáticas), a fin de agruparlos en una serie de categorías denominadas *clases de complejidad*.

La complejidad de un algoritmo viene dada por el número de operaciones elementales (sumas, productos, asignaciones, etc) requeridas en su ejecución, que depende del tamaño de las entradas. Se considera a su vez que la complejidad de un problema viene dada por la del algoritmo de menor complejidad que lo resuelve (en el caso del TSP, que garantice encontrar la ruta más corta).

El enfoque que se describe a lo largo de esta subsección se basa en la construcción teórica dada en Korte y Vygen [2006], que parte del concepto de *máquina de Turing* (ver Definición 1.10) sobre el cual se definen y construyen una serie de resultados propios de la teoría de la complejidad computacional. Para introducir este término es necesario definir previamente lo que se entiende en este contexto por *alfabeto*.

Definición 1.9. *Un* ***alfabeto*** $\mathcal{A}$ *es un conjunto finito de al menos dos elementos (salvo el símbolo especial* $\sqcup$*, que se usa para representar el espacio en blanco). Se denota por* $\mathcal{A}^n$ *al conjunto de todas las cadenas o secuencias de elementos de* $\mathcal{A}$ *que tienen longitud* n *(formadas por* n *elementos no necesariamente diferentes), por* $\mathcal{A}^* = \cup_{n\in\mathbb{Z}_+}\mathcal{A}^n$*, al conjunto de todos las cadenas (finitas) sobre* $\mathcal{A}$ *y por* $\mathcal{A}^0$ *al conjunto que únicamente contiene la cadena vacía. Cualquier subconjunto* $L \subset \mathcal{A}^*$ *recibe el nombre de* ***lenguaje sobre el alfabeto*** $\mathcal{A}$ *y cada uno de sus elementos recibe el nombre de* ***palabra*** *sobre* L*.*

Notación. *Sea* x *una cadena sobre un alfabeto* $\mathcal{A}$*, se denota por* $tam(x)$ *a la longitud de la cadena* x*, es decir,* $tam(x) = n \Leftrightarrow x \in \mathcal{A}^n$*.*

Comúnmente se trabaja con el alfabeto binario $\mathcal{A} = \{0, 1\}$ junto al lenguaje L de todas las cadenas finitas de ceros y unos (i.e. $L = \mathcal{A}^*$).

La Definición 1.10 contiene el concepto de *máquina de Turing* cuyo origen se atribuye al matemático inglés Alan Turing, que la da nombre.

Definición 1.10 (Turing, 1936)**.** *Sea $\mathcal{A}$ un alfabeto y $\overline{\mathcal{A}} = \mathcal{A} \cup \{\sqcup\}$. Una* ***máquina de Turing*** *sobre el alfabeto $\mathcal{A}$ viene dada por una función*

$$\Phi : \{0, 1, \ldots, N\} \times \overline{\mathcal{A}} \longrightarrow \{-1, \ldots, N\} \times \overline{\mathcal{A}} \times \{-1, 0, 1\},$$

para cierto $N \in \mathbb{Z}_+$. El ***procesamiento*** *de la cadena $x \in \mathcal{A}^*$ por Φ es la secuencia finita o infinita de 3-uplas de la forma $(n^{(i)}, s^{(i)}, \pi^{(i)})$ con $n^{(i)} \in \{-1, \ldots, N\}$, $s^{(i)} = (\ldots, s^{(i)}_{-1}, s^{(i)}_0, s^{(i)}_1, \ldots, s^{(i)}_j, \ldots) \in \overline{\mathcal{A}}^{\mathbb{Z}}$ y $\pi^{(i)} \in \mathbb{Z}$ (donde i indica la iteración, $i = 0, 1, 2, \ldots$) definidos recursivamente de la forma siguiente:*

- *Para $i = 0$,*

$$n^{(0)} := 0;\ \pi^{(0)} := 1;\ s^{(0)}_j = \begin{cases} x_j & si \quad 1 \leq j \leq tam(x) \\ \sqcup & si \quad j \leq 0 \ y \ j > tam(x) \end{cases}$$

- *Para cada $i > 0$:*
  - *Si $n^{(i)} \neq -1$ entonces*

$$\begin{aligned} (m, \sigma, \delta) &:= \Phi(n^{(i)}, s^{(i)}_{\pi^{(i)}}); \\ n^{(i+1)} &:= m; \\ s^{(i+1)}_j &:= \begin{cases} \sigma & si \quad j = \pi^{(i)}; \\ s^{(i)}_j & si \quad j \in \mathbb{Z} \setminus \{\pi^{(i)}\}; \end{cases} \\ \pi^{(i+1)} &:= \pi^{(i)} + \delta. \end{aligned}$$

  - *Si $n^{(i)} = -1$: se ha llegado a la última iteración del proceso, por lo que se dice que x se ha procesado en i iteraciones, denotado por $tiempo(\Phi, x) := i$, obteniendo como salida una nueva cadena en A^k con*

$$k = min\{j \in \mathbb{N} : s^{(i)}_j = \sqcup\} - 1,$$

que se denota por $salida(\Phi, x)$, dada por

$$salida(\Phi, x)_j := s^{(i)}_j, \quad para\ cada\ j\ en\ 1 \leq j \leq k.$$

En el caso de producirse un bucle infinito (i.e. $n^{(i)} \neq -1$ para todo $i \geq 0$) se considera $tiempo(\Phi, x) := \infty$ y $salida(\Phi, x)$ como indefinida.

De manera informal, puede decirse que una máquina de Turing sobre un alfabeto $\mathcal{A}$, es aquella que puede procesar cadenas de elementos del alfabeto a través de un componente que se desplaza por la cadena a modo de cabezal lectura-escritura. Las operaciones que realiza, así como su desplazamiento a lo largo de la cadena, están predeterminados.

Una máquina de Turing se caracteriza por

- un conjunto de $N+2$ estados, donde N es un entero no negativo, que se denotan por n, entero variando entre -1 y N y
- una tabla de procesamiento interna, que dado un estado n y un elemento a del alfabeto, devuelve tres componentes: un nuevo estado de la máquina n', un nuevo elemento b del alfabeto y un valor de desplazamiento. En las condiciones de la Definición 1.10, esta correspondencia vendría dada por la aplicación Φ.

La máquina recibe como entrada una cadena $x \in \mathcal{A}^*$, que se completa con infinitos espacios en blanco (simbolizados por $\sqcup$) a ambos lados. Esta cadena es procesada mediante un procedimiento iterativo. Partiendo del estado 0 y del primer elemento de la cadena, se toma cada vez un elemento y el estado actual de la máquina como entrada para la tabla de procesamiento, que devuelve a su vez el nuevo elemento que pasará a ocupar la posición que ocupaba el elemento actual en la cadena (lo sobrescribe), así como el nuevo estado de la máquina y el desplazamiento. Este último valor indica el elemento que será procesado en la siguiente iteración, que puede ser aquel que sobrescribe al anterior (valor 0), el situado inmediatamente a la izquierda (valor -1) o a la derecha (valor 1). El estado -1 indica el fin del procesamiento.

Ejemplo 1.11. *La siguiente máquina de Turing sobre el alfabeto $\mathcal{A} = \{0,1\}$ obtiene como salida el complemento a 1 de la cadena binaria introducida.*

$$\begin{array}{rcl} \Phi : \{0\} \times \overline{\mathcal{A}} & \longrightarrow & \{-1,0\} \times \overline{\mathcal{A}} \times \{-1,0,1\} \\ \Phi(0,0) & = & (0,1,1); \\ \Phi(0,1) & = & (0,0,1); \\ \Phi(0,\sqcup) & = & (-1,\sqcup,0); \end{array}$$

Así, el procesamiento de la cadena $x = 1101 \in \mathcal{A}^4$ se ilustra de la forma siguiente:

$$\begin{array}{ccccccccc} \cdots & \overbrace{1} & 1 & 0 & 1 & \cdots & & (0,1) & \mapsto & (0,0,1) \\ \cdots & 0 & \overbrace{1} & 0 & 1 & \cdots & & (0,1) & \mapsto & (0,0,1) \\ \cdots & 0 & 0 & \overbrace{0} & 1 & \cdots & & (0,0) & \mapsto & (0,1,1) \\ \cdots & 0 & 0 & 1 & \overbrace{1} & \cdots & & (0,1) & \mapsto & (0,0,1) \\ \cdots & 0 & 0 & 1 & 0 & \overbrace{\sqcup} & \cdots & (0,\sqcup) & \mapsto & (-1,\sqcup,0) \\ \cdots & 0 & 0 & 1 & 0 & \cdots & & & & \end{array}$$

dando como resultado $salida(\Phi, x) = 0010 \in \mathcal{A}^4$ y $tiempo(\Phi, x) = 5$.

En las siguientes definiciones se introduce la terminología básica de la sección.

Definición 1.12. *Sea $\mathcal{A}$ un alfabeto, $S, T \subseteq \mathcal{A}^*$ dos lenguajes, $f : S \longrightarrow T$ una función y Φ una máquina de Turing sobre $\mathcal{A}$ tal que $\forall s \in S$*

$$tiempo(\Phi, s) < \infty \ \ y \ \ salida(\Phi, s) = f(s).$$

*Se dice entonces que f **es procesable por** Φ o que Φ **procesa a** f. Si además existe un polinomio p tal que para cada $s \in S$ se cumple que*

$$tiempo(\Phi, s) \leq p(tam(s))$$

entonces se dice que la máquina de Turing Φ *tiene* ***complejidad computacional de orden polinomial*** *o que* Φ ***es polinomial***.

La Definición 1.13 no es más que un caso concreto de la definición anterior y tiene una importante relación con un tipo fundamental de problemas que veremos a continuación: los *problemas de decisión*.

Definición 1.13. *Sea* $\mathcal{A}$ *un alfabeto tal que* $0, 1 \in \mathcal{A}$*, sean* $S = \mathcal{A}^*$ *y* $T = \{0, 1\} \subset \mathcal{A}$ *dos lenguajes. Sea* Φ *una máquina de Turing y* $f : S \longrightarrow T$ *una función procesable por* Φ*. Se dice entonces que* Φ ***decide el lenguaje*** L*, dado por*

$$L := \{s \in S : f(s) = 1\}.$$

Recíprocamente, se dice que un lenguaje L ***es decidible*** *si existe una máquina de Turing* Φ *y una función* f *que sea procesable por* Φ*. Si además* Φ *tiene complejidad computacional de orden polinomial diremos que* L ***es decidible polinomialmente***.

Existen máquinas de Turing con una definición más compleja, y que sustentan algunos conceptos de la teoría de la complejidad computacional. La siguiente definición presenta un tipo específico de ellas, de utilidad para catalogar un conjunto de problemas entre los que está el TSP: los problemas $\mathcal{NP}$*-hard*.

Definición 1.14. *Sea* $\mathcal{A}$ *un alfabeto y* $\overline{\mathcal{A}} = \mathcal{A} \cup \{\sqcup\}$*. Sea* $X \subseteq \mathcal{A}^*$ *y sea* $f(x) \subseteq \mathcal{A}^*$ *un lenguaje no vacío para cada* $x \in X$*. Una* ***máquina oráculo que usa*** f *es una función*

$$\Phi : \{0, \ldots, N\} \times \overline{\mathcal{A}}^2 \longrightarrow \{-2, \ldots, N\} \times \overline{\mathcal{A}}^2 \times \{-1, 0, 1\}^2$$

para cierto $N \in \mathbb{Z}_+$*, cuyo procesamiento de una entrada* $x \in \mathcal{A}^*$ *es la secuencia (finita o infinita) de 5-uplas de la forma*

$$(n^{(i)}, s^{(i)}, t^{(i)}, \pi^{(i)}, \rho^{(i)}), \quad con\ n^{(i)} \in \{-2, \ldots, N\},\ s^{(i)}, t^{(i)} \in \overline{\mathcal{A}}^{\mathbb{Z}}\ \ y\ \ \pi^{(i)}, \rho^{(i)} \in \mathbb{Z} \tag{1.3.1}$$

con $i = 0, 1, 2, \ldots$*. Este procesamiento viene dado de la forma siguiente:*

- *Para* $i = 0$ *se tiene que*

$$n^{(0)} := 0;\ s_j^{(0)} := \begin{cases} x_j & si \quad 1 \leq j \leq tam(x); \\ \sqcup & si \quad j \leq 0\ y\ j > tam(x); \end{cases} \quad t_j^{(0)} := \sqcup,\ \forall j \in \mathbb{Z};\ \pi^{(0)}, \rho^{(0)} := 1. \tag{1.3.2}$$

- *Para* $i > 0$*, se distinguen tres casos:*

- *Si* $n^{(i)} \neq -1$*, entonces se asocia*

$$\begin{aligned}
(m, \sigma, \tau, \delta, \epsilon) &:= \Phi(n^{(i)}, s^{(i)}_{\pi^{(i)}}, t^{(i)}_{\rho^{(i)}}); \\
n^{(i+1)} &:= m; \\
s^{(i+1)}_j &:= \begin{cases} \sigma & si \quad j = \pi^{(i)}; \\ s^{(i)}_j & si \quad j \in \mathbb{Z} \setminus \{\pi^{(i)}\}; \end{cases} \\
t^{(i+1)}_j &:= \begin{cases} \tau & si \quad j = \rho^{(i)}; \\ t^{(i)}_j & si \quad j \in \mathbb{Z} \setminus \{\rho^{(i)}\}; \end{cases} \\
\pi^{(i+1)} &:= \pi^{(i)} + \delta; \\
\rho^{(i+1)} &:= \rho^{(i)} + \epsilon.
\end{aligned}$$

- *Si* $n^{(i)} = -1$*: se ha llegado a la última iteración del proceso, por tanto se dice que tiempo*$(\Phi, x) := i$ *y que obtiene salida*$(\Phi, x) = s^{(i)}_j$ *para todo* j *cumpliendo* $1 \leq j \leq k$ *y donde* $k := min\{j \in \mathbb{N} : s^{(i)}_j = \sqcup\} - 1$.
- *A mayores, si en alguna iteración* i *se obtiene*

$$\Phi(n^{(i)}, s^{(i)}_{\pi^{(i)}}, t^{(i)}_{\rho^{(i)}}) = (-2, \sigma, \tau, \delta, \epsilon),$$

se considera $k = \min\{j \in \mathbb{N} : t^{(i-1)}_j = \sqcup\} - 1$ *y* $x \in \mathcal{A}^k$ *dada por*

$$x_j = t^{(i)}_j, \text{ para cada } j \text{ con } 1 \leq j \leq tam(y).$$

Si $x \in X$*, se elige* $y \in f(x)$ *arbitrario y se sobrescribe*

$$t^{(i+1)}_j := \begin{cases} y_j & si \quad 1 \leq j \leq tam(y) \\ \sqcup & si \quad j = tam(y) + 1 \end{cases}$$

permaneciendo el resto de posiciones estables. El procedimiento continua con $n^{(i+1)} := n^{(i)} + 1$ *(deteniéndose si* $n^{(i)} = -1$*).*

Lo que diferencia este modelo con el presentado en la Definición 1.10, es que en las máquinas de Turing la salida es siempre idéntica para una misma entrada, mientras que para una máquina oráculo pueden darse resultados distintos en varios procesamientos sobre la misma entrada.

La utilidad más importante de ambos conceptos es el modelado de cualquier algoritmo orientado a la resolución de problemas computacionales, donde la entrada representa la instancia a resolver y la salida no es más que la solución encontrada. Así, se dice que un algoritmo tiene complejidad de orden **polinomial** (o simplemente que es polinomial) si existe una máquina que lo modele cuyo procesamiento tenga una complejidad computacional asociada de orden polinomial. A su vez, se dice que un algoritmo es *oráculo* cuando es modelable por una máquina oráculo.

A continuación se introducen los denominados *problemas de decisión*, que juegan un papel fundamental en el desarrollo de la teoría de la complejidad computacional.

Definición 1.15. *Un **problema de decisión** es un par $P = (X, Y)$ donde X es un lenguaje decidible polinomialmente e $Y \subseteq X$.*

De modo informal podemos decir que un problema de decisión es aquel que consiste en encontrar la respuesta a una pregunta que únicamente admite *sí/no* como respuesta.

Ejemplo 1.16. *Los dos problemas siguientes caracterizan a la programación lineal y a la programación lineal de enteros respectivamente.*

- *Sea M una matriz en $\mathbb{Z}^{mxn}$ y sea b un vector en $\mathbb{Z}^m$. El problema de determinar si existe un vector $x \in \mathbb{Q}^n$ tal que $Mx \leq b$ es un problema de decisión.*
- *En las mismas condiciones que el apartado anterior, el problema de determinar si existe un vector $x \in \mathbb{Z}^n$ tal que $Mx \leq b$ es un problema de decisión.*

En las condiciones de la Definición 1.15, X representa el conjunto de todas las instancias del problema P e Y aquellas instancias con respuesta afirmativa o *instancias afirmativas*. Por el contrario, las instancias en $X \setminus Y$ se denominan *instancias negativas*.

Definición 1.17. *Sea P un problema de decisión. Se dice que P **es de clase** $\mathcal{P}$ o que $P \in \mathcal{P}$ si existe un algoritmo polinomial que lo resuelve.*

No todos los problemas de decisión pertenecen a $\mathcal{P}$. Existen algunos cuyos algoritmos pueden llegar a arrojar la solución, en el peor caso, con un coste computacional de orden exponencial, por lo que ante instancias grandes el tiempo de ejecución se eleva en exceso.

Existen a su vez otro tipo de algoritmos que se limitan a comprobar un resumen de la instancia o *certificado*, en lugar de tratar con la instancia completa. Los problemas que admiten algoritmos polinomiales de este tipo se dice que son de clase $\mathcal{NP}$.

Definición 1.18. *Se denota por $\mathcal{NP}$ (non-deterministic polynomial time problems) a la clase de los problemas de decisión tales que, si $P = (X, Y) \in \mathcal{NP}$, existe un polinomio p y otro problema de decisión $P' = (X', Y') \in \mathcal{P}$ tal que*

$$X' := \{x\#c : x \in X,\, c \in \{0,1\}^{\lfloor p(tam(x))\rfloor}\} \tag{1.3.3}$$

y además

$$Y = \{y \in X : \exists c \in \{0,1\}^{\lfloor p(tam(y))\rfloor} \mid y\#c \in Y'\}, \tag{1.3.4}$$

*donde $x\#c$ es la concatenación de las cadenas x y c unidas por el símbolo especial $\#$. En estas condiciones, c se dice que es el **certificado** de la instancia x. Cualquier algoritmo que resuelva el problema P' recibe el nombre de **algoritmo de comprobación de certificados** (certificated-checking algorithm).*

Un algoritmo basado en la comprobación de certificados se limita por tanto a decidir mediante el certificado si la instancia a la que pertenece es afirmativa o negativa, por lo

que el coste computacional del procesamiento será polinomial si así lo es el tamaño del certificado, como fija la Definición 1.18.

Cualquier problema de clase $\mathcal{P}$ puede resolverse mediante un algoritmo polinomial de comprobación de certificados, pues todo algoritmo puede considerarse de este tipo añadiendo un certificado vacío a cada instancia. El siguiente resultado prueba esta afirmación.

Proposición 1.19. $\mathcal{P} \subseteq \mathcal{NP}$.

Demostración. Sea $P = (X, Y) \in \mathcal{P}$ y sea p el polinomio nulo, se construye el problema $P' = (X', Y')$ donde

$$X' = \{x\#c : x \in X, c \in \{0,1\}^0\} = \{x\# : x \in X\}.$$

Por tanto, existirá un algoritmo para la resolución de P' cuyo procesamiento consista, primero, en eliminar el símbolo # y después ejecutar un algoritmo polinomial de resolución de P aplicado a la instancia $x \in P$ resultante. El nuevo algoritmo será por tanto polinomial, concluyéndose que $P \in \mathcal{NP}$. □

Un problema pueden modelarse en base a otro, de manera que un algoritmo que resuelva el segundo también aplica al primero. Esta es la idea detrás del concepto de *reducción.*

Definición 1.20. *Sean P_1 y $P_2 = (X, Y)$ dos problemas de decisión. Sea $f : X \longrightarrow \{0,1\}$ dada por $f(x) = 1$ para cada $x \in Y$ y $f(x) = 0$ para cada $x \in X \setminus Y$. Se dice que P_1* ***reduce polinomialmente en*** *P_2 si existe un algoritmo oráculo polinomial para P_1 que use f.*

El siguiente resultado da sentido al concepto anterior.

Proposición 1.21. *Si P_1 reduce polinomialmente en P_2 y existe un algoritmo oráculo para P_2, entonces existe un algoritmo polinomial para P_1.*

Demostración. Sea $P_1 = (X_1, Y_1)$ y $P_2 = (X_2, Y_2)$. Sea A_2 un algoritmo para P_2 tal que $tiempo(A_2, x) \leq p_2(tam(x))$ para cada $x \in X_2$ y sea $f : X_2 \longrightarrow \{0,1\}$ la función dada por $f(x) = salida(A_2, x)$. Sea A_1 un algoritmo oráculo para P_1 que use f con $tiempo(A_1, x) \leq p_1(tam(x))$ para cada $x \in X_1$. Entonces reemplazando el procedimiento del algoritmo oráculo A_1 por otro basado en A_2, se obtiene un algoritmo A_3 para P_1 tal que, para cada $x \in X_1$ se tiene que $tiempo(A_3, x) \leq p_1(tam(x)) \cdot p_2(p_1(tam(x)))$, ya que a lo sumo se producirían $p_1(tam(x))$ iteraciones del algoritmo y ninguna de las instancias que procesará será mayor que $p_1(tam(x))$. Al ser p_1 y p_2 polinomios, se concluye que A_3 es polinomial. □

Se introduce ahora el concepto de problema $\mathcal{NP}$*-completo*, para lo cual se expone previamente el concepto de *transformación*, que no es más que una variante de la reducción.

Definición 1.22. *Sean $P_1 = (X_1, Y_1)$ y $P_2 = (X_2, Y_2)$ dos problemas de decisión. Se dice que P_1 se* ***transforma polinomialmente*** *en P_2 si existe una función $f : X_1 \to X_2$ y un*

algoritmo polinomial que la procesa tal que $f(y_1) \in Y_2$ *para todo* $y_1 \in Y_1$ *y* $f(x_1) \in X_2 \setminus Y_2$ *para todo* $x_1 \in X_1 \setminus Y_1$.

Por tanto, decir que un problema se transforma en otro quiere decir que cada instancia afirmativa (resp. negativa) del primero se corresponde con una instancia afirmativa (resp. negativa) del segundo, y que el coste computacional de esta asociación es polinomial.

Definición 1.23. *Se dice que un problema de decisión* $P \in \mathcal{NP}$ *es* $\mathcal{NP}$***-completo*** *si todo problema de la clase* $\mathcal{NP}$ *se transforma en* P.

Obsérvese que, si se encuentra un algoritmo polinomial que resuelva un problema $\mathcal{NP}$*-completo*, existiría entonces un algoritmo polinomial para cada problema $P \in \mathcal{NP}$ que lo resolviese pudiendo concluir por la Proposición 1.19 que $\mathcal{P} = \mathcal{NP}$.

Por último se introduce otro tipo de problemas, cuyo objetivo ya no es determinar si la respuesta a una pregunta planteada es afirmativa o negativa como en los problemas de decisión, sino encontrar una solución que minimice o maximice el valor de una función. Estos son los denominados *problemas de optimización* que se definen a continuación.

Definición 1.24. *Un* ***problema de optimización (discreto)*** *es una 4-tupla*

$$P = (X, (S_x)_{x \in X}, c, obj) \tag{1.3.5}$$

donde

- X *es un lenguaje sobre el alfabeto* $\{0,1\}$ *decidible polinomialmente,*
- S_x *es un subconjunto de* $\{0,1\}^*$ *tal que existe un polinomio* p *con* $tam(y) \leq p(tam(x))$ *para todo* $y \in S_x$ *y para todo* $x \in X$. *Además los lenguajes* $\{(x,y) : x \in X,\, y \in S_x\}$ *y* $\{x \in X : S_x = \varnothing\}$ *son decidibles polinomialmente,*
- $c : \{(x,y) : x \in X,\, y \in S_x\} \to \mathbb{Q}$ *es una función procesable en tiempo polinomial,*
- $obj \in \{\text{máx}, \text{mín}\}$ *(según el objetivo sea maximizar o minimizar la función* c*).*

Los elementos de X *se denominan* ***instancias*** *de* P, *y para cada instancia* $x \in X$, *los elementos de* S_x *reciben el nombre de* ***soluciones factibles de*** x *(o simplemente soluciones de* x*). De entre ellas, una* ***solución óptima*** *será aquella para la que:*

$$c(x,y) = obj\{c(x,z) : z \in S_x\} \tag{1.3.6}$$

Se denota como $OPT(x) := obj\{c(x,z) : z \in S_x\}$.

Un ***algoritmo*** *para un problema de optimización* $(X, (S_x)_{x \in X}, c, obj)$ *es aquel que para cada entrada* $x \in X$ *con* $S_x \neq \varnothing$ *devuelve una solución factible* $y \in S_x$. *Decimos además que es* ***exacto*** *si la solución que devuelve para cada* $x \in X$ *es además óptima.*

La función $c(x,y)$ recibe el nombre de *función objetivo*, que por ejemplo, para el TSP, representa la longitud de la ruta y, solución factible de la instancia x.

Definición 1.25. *Un problema de optimización* P *se dice que es de clase* $\mathcal{NP} - hard$ *si cualquier problema de clase* $\mathcal{NP}$ *reduce polinomialmente en* P.

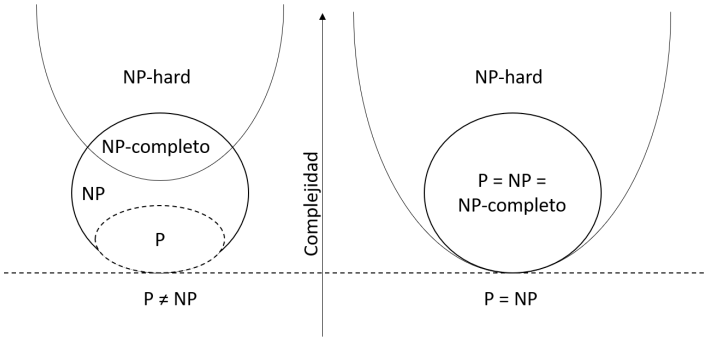


Figura 1.2: Modelo representativo de las clases de complejidad de los problemas.

La Figura 1.2 ilustra la relación existente entre las diferentes clases de problemas distinguiendo entre si $\mathcal{P}$ y $\mathcal{NP}$ son, o no, el mismo conjunto.

A continuación se recoge el concepto de *restricción* de un problema.

Definición 1.26. *Dado un problema de optimización* $P = (X, (S_x)_{x\in X}, c, obj)$ *y* $X' \subseteq X$, *se dice que el problema* $P' = (X', (S_x)_{x\in X'}, c, obj)$ ***es una restricción de*** P.

En base a la Definición 1.26 puede considerarse un subconjunto de los problemas $\mathcal{NP}$*-hard*, entre los que se encuentra el TSP, para los cuales existe una restricción que es a su vez $\mathcal{NP}$*-hard*. Estos problemas reciben el nombre de problemas fuertemente $\mathcal{NP}$*-hard*.

Definición 1.27. *Sea* $P = (X, (S_x)_{x\in X}, c, obj)$ *un problema de optimización sobre el alfabeto* $\mathcal{A} = \mathbb{Z}$. *Dado un polinomio* p *se denota por* $P_p = (X_p, Y_p)$ *a una restricción de* P *dada por*

$$X_p = \{x \in X : \text{máx}(x) \leq p(tam(x))\}.$$

Se dice entonces que P ***es fuertemente*** $\mathcal{NP}$***-hard*** *si existe un polinomio* p *tal que* P_p *es* $\mathcal{NP}$*-hard.*

Las Definiciones 1.25, 1.26, y 1.27 abarcan también a los problemas de decisión pero no es objeto de este Trabajo Fin de Grado aplicar su contenido a este tipo de problemas.

1.3.2. Los Teoremas de Cook

Una vez han sido fijados los conceptos básicos de la teoría de la complejidad computacional, indagamos ahora en el trabajo que el matemático estadounidense Stephen Cook recogió en 1971 en su artículo *The Complexity of Theorem Proving Procedures*. Tomando algunos de sus resultados fundamentales como base, se prueba la ya mencionada complejidad del TSP, en el Corolario 1.38.

Esta sección parte del *problema de la satisfacibilidad booleana*, para lo cual se recogen una serie de conceptos previos en la Definición 1.28.

Definición 1.28. *Sea $X = \{x_1, x_2, \ldots, x_k\}$ un conjunto de* ***variables booleanas*** *(i.e. variables tales que $x_i \in \{true, false\}$, para todo $i = 1, 2, \ldots, k$). Un* ***valor de verdad*** *para X es una función $T : X \to \{true, false\}$. T es extendida a su vez al conjunto $L = X \cup \{\overline{x} : x \in X\}$ (donde $\overline{x}$ denota la negación lógica de x) de manera que*

$$T(\overline{x}) = \begin{cases} true & si \quad T(x) = false \\ \\ false & si \quad T(x) = true \end{cases}$$

Los elementos de L se denominan ***literales*** *sobre X.*

Se define una ***cláusula*** *sobre X como un conjunto de literales sobre X, y se dice que es* ***satisfecha*** *por T si al menos uno de sus literales es cierto para el valor de verdad dado. Una familia $\mathcal{Z}$ de cláusulas sobre X se dice que es* ***satisfacible*** *si existe un* ***valor de verdad*** *que simultáneamente satisface todas las cláusulas.*

Un literal, por tanto, no es más que una fórmula atómica, la cual puede estar compuesta de una variable booleana o de la negación de ésta.

Ejemplo 1.29. *Dado el conjunto de variables booleanas $X = \{x_1, x_2, x_3\}$, la cláusula $Z = \{x_1, \overline{x_2}, \overline{x_3}\}$ y el valor de verdad T para X dado por*

$$T(x_1) = true, \;\; T(x_2) = false \;\; y \;\; T(x_3) = true, \tag{1.3.7}$$

se obtienen los valores booleanos $\{true, true, false\}$ para la cláusula Z, por lo que T es un valor de verdad que satisface la cláusula Z.

En estas condiciones podemos plantear el *problema de la satisfacibilidad general.*

Definición 1.30. *Sea X un conjunto de variables booleanas y $\mathcal{Z}$ un conjunto de cláusulas sobre X. El* ***problema de la satisfacibilidad boolenana o satisfacibilidad general*** *consiste en determinar si $\mathcal{Z}$ es o no satisfacible.*

El objeto de presentación del problema de la satisfacibilidad general reside en su complejidad, base para la prueba del Teorema 1.35. Así, el Teorema 1.31 enuncia la ya mencionada complejidad de este problema.

Teorema 1.31 (Cook, 1971)**.** *El problema de la satisfacibilidad general (ver Definición 1.30) es $\mathcal{NP}$-completo.*

Demostración. Para probar este resultado basta con ver que el problema es $\mathcal{NP}$ y que cualquier otro problema P de clase $\mathcal{NP}$ transforma en el primero.

La primera parte es inmediata, pues basta tomar como certificado el valor de verdad y considerar un algoritmo que lo compruebe para concluir.

Para la segunda, se considera $P = (X, Y)$ otro problema en $\mathcal{NP}$. La Definición 1.18 fija la existencia de un polinomio p y un problema de decisión $P' = (X', Y')$ en $\mathcal{P}$ donde

$$X' = \{x \# c : x \in X, c \in \{0,1\}^{\lfloor p(tam(x)) \rfloor}\}$$

e

$$Y = \{y \in X : \exists c \in \{0,1\}^{\lfloor p(tam(x))\rfloor} \text{ con } y\#c \in Y'\}.$$

Sea $\mathcal{A}$ un alfabeto y $\Phi : \{0,\ldots,N\} \times \overline{\mathcal{A}} \to \{-1,\ldots,N\} \times \overline{\mathcal{A}} \times \{-1,0,1\}$ una máquina de Turing sobre $\mathcal{A}$ que procesa polinomialmente a P', siendo q el polinomio tal que $tiempo(\Phi, x\#c) \leq q(tam(x\#c))$ para todo $x\#c \in X'$, y donde $tam(x\#c) = tam(x) + 1 + \lfloor p(tam(x))\rfloor$.

Fijado $x \in X$, se considera el conjunto de variables booleanas $V(x)$ formado por

- $v_{ij\sigma}$ para todo $0 \leq i \leq Q_x$, $-Q_x \leq j \leq Q_x$ y $\sigma \in \overline{A}$,
- w_{ijn} para todo $0 \leq i \leq Q_x$, $-Q_x \leq j \leq Q_x$ y $-1 \leq n \leq N$.

Se construye una colección de cláusulas $\mathcal{Z}(x)$ sobre $V(x)$ que se relaciona de forma equivalente con Φ mediante un valor de verdad T sobre $V(x)$ dado por

$$T(v_{i,j,\sigma}) = \begin{cases} true \quad si \quad s_j^{(i)} = \sigma; \\ \\ false \quad si \quad no; \end{cases} \qquad T(w_{i,j,n}) = \begin{cases} true \quad si \quad \pi^{(i)} = j \text{ y } n^{(i)} = n; \\ \\ false \quad si \quad no; \end{cases}$$

$\mathcal{Z}(x)$ contiene las siguientes cláusulas agrupadas según las condiciones que modelan:

1. A cada instante, cada posición de la cadena contiene un único símbolo:
   - $\{v_{ij\sigma} : \sigma \in \overline{A}\}$ para $0 \leq i \leq Q_x$ y $-Q_x \leq j \leq Q_x$;
   - $\{\overline{v_{ij\sigma}}, \overline{v_{ij\tau}}\}$ para todo $0 \leq i \leq Q_x$, $-Q_x \leq j \leq Q_x$ y $\sigma, \tau \in \overline{\mathcal{A}}$ con $\sigma \neq \tau$.
2. A cada instante se escanea una única posición y se ejecuta una única instrucción:
   - $\{w_{ijn} : -Q_x \leq j \leq Q_x, -1 \leq n \leq N\}$ para todo $0 \leq i \leq Q_x$;
   - $\{\overline{w_{ijn}}, \overline{w_{ij'n'}}\}$ para todo $0 \leq i \leq Q_x$, $-Q_x \leq j, j' \leq Q_x$ y $-1 \leq n, n' \leq N$ con $j \neq j'$ y $n \neq n'$.
3. El algoritmo comienza correctamente con la entrada $x\#c$ para algún $c \in \{0,1\}^{\lfloor p(tam(x))\rfloor}$ y donde $x = (x_1, x_2, \ldots, x_{tam(x)})$:
   - $\{v_{0,j,x_j}\}$ para $1 \leq j \leq tam(x)$;
   - $\{v_{0.tam(x)+1,\#}\}$;
   - $\{v_{0,tam(x)+1+j,0}, v_{0,tam(x)+1+j,1}\}$ para $1 \leq j \leq \lfloor p(tam(x))\rfloor$;
   - $\{v_{0,j,\sqcup}\}$ para $-Q_x \leq j \leq 0$ y $tam(x) + 2 + \lfloor p(tam(x))\rfloor \leq j \leq Q_x$;
   - $\{w_{0,1,0}\}$.
4. El algoritmo funciona correctamente:
   - $\{\overline{v_{ij\sigma}}, \overline{w_{ijn}}, v_{i+1,j,\tau}\}, \{\overline{v_{ij\sigma}}, \overline{w_{ijn}}, w_{i+1,j+\delta,m}\}$ para $0 \leq i \leq Q_x$, $-Q_x \leq j \leq Q_x$, $\sigma \in \overline{\mathcal{A}}$ y $0 \leq n \leq N$ donde $\Phi(n, \sigma) = (m, \tau, \delta)$.
5. Cuando el algoritmo alcanza el valor -1, se detiene:
   - $\{\overline{w_{i,j,-1}}, w_{i+1,j,-1}\}, \{\overline{w_{i,j,-1}}, \overline{v_{i,j,\sigma}}, v_{i+1,j,\sigma}\}$ para $0 \leq i < Q_x, -Q_x \leq j \leq Q_x$ y $\sigma \in \overline{\mathcal{A}}$.

6. Las posiciones sin escanear permanecen intactas:
   - $\{\overline{v_{ij\sigma}}, \overline{w_{ij'n}}, v_{i+1,j,\sigma}\}$ para $0 \leq i \leq Q_x, \sigma \in \overline{\mathcal{A}}, -1 \leq n \leq N$ y $-Q_x \leq j, j' \leq Q_x$ con $j \neq j'$.
7. La salida del algoritmo es 1:
   - $\{v_{Q,1,1}\}, \{v_{Q,2,\sqcup}\}$.

Sobre la construcción anterior se prueba que $\mathcal{Z}(x)$ es satisfacible si, y sólo si, $x \in Y$.

Si $Z(x)$ es satisfacible, existe un valor de verdad T tal que satisface todas las cláusulas. Sea entonces $c \in \{0,1\}^{\lfloor p(tam(x)) \rfloor}$ con $c_j = 1$ para todo j con $T(v_{0,tam(x)+1+j,1}) = true$ y $c_j = 0$ en caso contrario. Por la construcción anterior, las variables reflejan el cálculo de la entrada $x\#c$. Por tanto, es posible concluir que $salida(\Phi, x\#c) = 1$. Como Φ es un algoritmo de comprobación de certificados, se deduce que x es una instancia afirmativa.

Recíprocamente, si $x \in Y$, sea c un certificado para x y sea $(n^{(i)}, s^{(i)}, \pi^{(i)})_{i=0,1,\ldots,m}$ el procesamiento de Φ con entrada $x\#c$. Se considera el valor de verdad T y se asocia, para $i = m+1, \ldots, Q_x$, $T(v_{i,j,\sigma}) = T(v_{i-1,j,\sigma})$ y $T(w_{i,j,n}) = T(w_{i-1,j,n})$ para todo j, n, σ. Entonces T hace satisfacible a $\mathcal{Z}(x)$.

Visto que existe la transformación del problema P en el de satisfacibilidad general, falta ver que es polinomial. Basta observar que dado $x \in X$, el algoritmo que construye la familia de cláusulas $\mathcal{Z}(x)$ tiene complejidad $\mathcal{O}(Q_x^3 log\, Q_x)$, por ser $\mathcal{O}(Q_x^3)$ el número de literales en las cláusulas, que ocupan $\mathcal{O}(log\, Q_x)$ cada uno. Como Q_x es un polinomio respecto de p,Φ y q, entradas del algoritmo, se concluye que esta transformación es polinomial. □

Vista la complejidad del problema de satisfacibilidad general, en el Teorema 1.35 se prueba la complejidad del problema restringido a cláusulas con exactamente 3 literales.

Definición 1.32. *Sea X un conjunto de variables booleanas y $\mathcal{Z}$ un conjunto de cláusulas sobre X con exactamente 3 literales cada una. El* ***problema de la satisfacibilidad booleana restringido a 3 literales (3-SAT)*** *consiste en determinar si $\mathcal{Z}$ es satisfacible.*

El siguiente resultado prueba que cualquier cláusula es equivalente a un conjunto de cláusulas con exactamente tres literales, y tiene como objetivo la prueba del Teorema 1.35.

Proposición 1.33. *Sea X un conjunto de variables booleanas y Z una cláusula sobre X con $k \geq 1$ literales. Entonces, existe un conjunto Y de a lo sumo* $\max\{k-3, 2\}$ *variables, no necesariamente en X, y una familia $\mathcal{Z}'$ de a lo sumo* $\max\{k-2, 4\}$ *cláusulas sobre $X \cup Y$ tal que cada elemento de $\mathcal{Z}'$ tiene exactamente tres literales. Además, para cada familia $\mathcal{W}$ de cláusulas sobre X, se tiene que $\mathcal{W} \cup \{Z\}$ es satisfacible si, y sólo si, $\mathcal{W} \cup \mathcal{Z}'$ es satisfacible. Finalmente, $\mathcal{Z}'$ puede ser procesada con complejidad de orden $\mathcal{O}(k)$.*

Demostración. Se distinguen tres casos según el número de literales de Z:

- Si $Z = \{\lambda_1, \lambda_2, \lambda_3\}$, se concluye $\mathcal{Z}' = \{Z\}$.

- Si $Z = \{\lambda_1, \ldots, \lambda_k\}$, se escoge un conjunto $Y = \{y_1, \ldots, y_{k-3}\}$ de $k-3$ nuevas variables y se asocia

$$\mathcal{Z}' = \{\{\lambda_1, \lambda_2, y_1\}, \{\overline{y_1}, \lambda_3, y_2\}, \{\overline{y_2}, \lambda_4, y_3\}, \ldots, \{\overline{y_{k-4}}, \lambda_{k-2}, y_{k-3}\}, \{\overline{y_{k-3}}, \lambda_{k-1}, \lambda_k\}\}\,.$$

- Si $Z = \{\lambda_1, \lambda_2\}$ se elige $Y = \{y_1\}$ y el conjunto

$$\mathcal{Z}' = \{\{\lambda_1, \lambda_2, y_1\}, \{\lambda_1, \lambda_2, \overline{y_1}\}\}\,.$$

- Si $Z = \{\lambda_1\}$, se elige $Y = \{y_1, y_2\}$ y se asocia

$$\mathcal{Z}' = \{\{\lambda_1, y_1, y_2\}, \{\lambda_1, y_1, \overline{y_2}\}, \{\lambda_1, \overline{y_1}, y_2\}, \{\lambda_1, \overline{y_1}, \overline{y_2}\}\}\,.$$

Es trivial ver que Z es equivalente a $\mathcal{Z}'$, ya que en cada uno de los casos anteriores si un cierto valor de verdad T satisface Z también lo hará sobre todas las cláusulas de $\mathcal{Z}'$, y, por contrarrecíproco, si T no satisface Z tampoco lo hará con la familia $\mathcal{Z}'$.

Por último, es claro que la familia $\mathcal{Z}'$ puede procesarse con complejidad $\mathcal{O}(k)$, pues el número de cláusulas de $\mathcal{Z}'$ es $\mathcal{O}(k)$, siendo k el número de elementos de Z, y cada cláusula puede ser procesada con complejidad constante. □

Ejemplo 1.34. *Para ilustrar el resultado anterior, se construye una máquina de Turing*

$$\begin{array}{rcl} \Phi : \{0,1,\ldots,7\} \times \{0,1,\sqcup\} & \longrightarrow & \{-1,0,1\ldots,7\} \times \{0,1,\sqcup\} \times \{-1,0,1\} \\ (n,x) & \longmapsto & (m,\sigma,\delta) \end{array}$$

cuya definición viene dada por la Tabla 1.1 y su funcionamiento se ilustra con el siguiente ejemplo sencillo.

Sea $\mathcal{Z}'$ una familia de dos cláusulas donde, para un cierto valor de verdad,

$$\mathcal{Z}' = \{\{false, true, false\}, \{true, false, false\}\}$$

i) Se traducen los valores lógicos a binarios identificando true con el valor 1 y false con el valor 0, quedando

$$\mathcal{Z}' = \{\{0,1,0\}, \{1,0,0\}\}\,.$$

La máquina de Turing procesará la entrada 010100*.*

	n = 0	n = 1	n = 2	n = 3
x = 0	(1,0,1)	(2,0,1)	(3,0,1)	(3,0,3)
x = 1	(0,1,3)	(0,1,2)	(0,1,1)	(3,1,3)
x = ⊔	(4,⊔,-1)	(4,⊔,-1)	(4,⊔,-1)	(5,⊔,-1)
	n = 4	n = 5	n = 6	n = 7
x = 0	(4,⊔,-1)	(5,⊔,-1)	(-1,1,0)	(-1,0,0)
x = 1	(4,⊔,-1)	(5,⊔,-1)	(-1,1,0)	(-1,0,0)
x = ⊔	(6,⊔,1)	(7,⊔,1)	(-1,1,0)	(-1,0,0)

Tabla 1.1: Definición de Φ, en el contexto de la demostración de la Proposición 1.33

ii) El procesamiento de la cadena 010100 *por parte de* Φ *es el siguiente:*

$$
\begin{array}{cccccccccccc}
\cdots & \overset{\frown}{0} & 1 & 0 & 1 & 0 & 0 & \cdots & & (0,0) & \mapsto & (1,0,1) \\
\cdots & 0 & \overset{\frown}{1} & 0 & 1 & 0 & 0 & \cdots & & (1,1) & \mapsto & (0,1,2) \\
\cdots & 0 & 1 & 0 & \overset{\frown}{1} & 0 & 0 & \cdots & & (0,1) & \mapsto & (0,1,3) \\
\cdots & 0 & 1 & 0 & 1 & 0 & 0 & \overset{\frown}{\sqcup} & \cdots & (0,\sqcup) & \mapsto & (4,\sqcup,-1) \\
\cdots & 0 & 1 & 0 & 1 & 0 & \overset{\frown}{0} & \cdots & & (4,0) & \mapsto & (4,\sqcup,-1) \\
\cdots & 0 & 1 & 0 & 1 & \overset{\frown}{0} & \sqcup & \cdots & & (4,0) & \mapsto & (4,\sqcup,-1) \\
\cdots & 0 & 1 & 0 & \overset{\frown}{1} & \sqcup & \sqcup & \cdots & & (4,1) & \mapsto & (4,\sqcup,-1) \\
\cdots & 0 & 1 & \overset{\frown}{0} & \sqcup & \sqcup & \sqcup & \cdots & & (4,0) & \mapsto & (4,\sqcup,-1) \\
\cdots & 0 & \overset{\frown}{1} & \sqcup & \sqcup & \sqcup & \sqcup & \cdots & & (4,1) & \mapsto & (4,\sqcup,-1) \\
\cdots & \overset{\frown}{0} & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \cdots & & (4,0) & \mapsto & (4,\sqcup,-1) \\
\cdots & \overset{\frown}{\sqcup} & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \cdots & & (4,\sqcup) & \mapsto & (6,\sqcup,1) \\
\cdots & \overset{\frown}{\sqcup} & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \cdots & & (6,\sqcup) & \mapsto & (-1,1,0) \\
\cdots & 1 & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \cdots & & & &
\end{array}
$$

Por tanto, $salida(\Phi, 010100) = 1$, $tiempo(\Phi, 010100) = 12$.

De forma general, es fácil comprobar que, si t *es el número de cláusulas de* $\mathcal{Z}'$ *y* s *la traducción a cadena binaria, como se hizo en el ejemplo en i), de* $\mathcal{Z}'$*, entonces* $tiempo(\Phi, s) \leq 6t + 3$*. Por otro lado, hemos visto que, siendo* k *el número de literales de* Z*,* $t \leq \max\{k-2, 4\} \leq k + 3$*. Por tanto,*

$$tiempo(\Phi, s) \leq 6t + 3 \leq 6(k + 3) + 3 = 6k + 21,$$

y se concluye que, $tiempo(\Phi, s)$ *es* $\mathcal{O}(k)$*.*

La Proposición 1.33 permite establecer la prueba del siguiente teorema.

Teorema 1.35 (Cook, 1971)**.** *El problema 3-SAT es $\mathcal{NP}$-completo.*

Demostración. Es claro que por ser $\mathcal{NP}$ el problema de Satisfacibilidad general (ver Definición 1.30), 3-SAT también lo es, por lo que basta probar que el problema de Satisfacibilidad general se transforma en el problema restringido a tres variables para ver que cualquier problema $\mathcal{NP}$ transforma en 3-SAT. Se considera un conjunto $\mathcal{Z} = \{Z_1, \ldots, Z_m\}$ de cláusulas y se construye uno nuevo $\mathcal{Z}'$ de cláusulas con tres literales cada una de la forma siguiente: se reemplaza cada cláusula Z_i por un conjunto equivalente de cláusulas con tres literales, que sabemos que existe en virtud de la Proposición 1.33. □

Pasamos ahora a trabajar sobre el *problema de los ciclos hamiltonianos*, de cuya complejidad se deriva, como veremos a continuación, la ya enunciada complejidad del TSP.

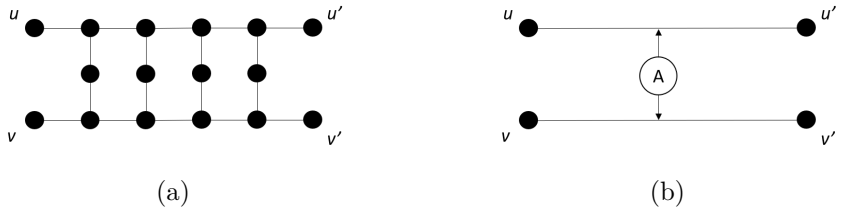
Definición 1.36. *Sea G un grafo. El* ***problema de los ciclos hamiltonianos*** *consiste en determinar si G es o no hamiltoniano*

Recordemos que un grafo G se dice que es hamiltoniano si admite un ciclo hamiltoniano (ver Definición 1.4) sobre él. Este problema es también $\mathcal{NP}$*-completo.*

Teorema 1.37. *El problema de los ciclos hamiltonianos es $\mathcal{NP}$-completo.*

Demostración. Probar que el problema es $\mathcal{NP}$ es trivial, pues un ciclo hamiltoniano sobre el grafo serviría como certificado, cuya comprobación tiene una complejidad asociada de orden polinomial. Por tanto, basta ver que 3-SAT se transforma el problema de los ciclos hamiltonianos para concluir la prueba.

Sea $\mathcal{Z} = \{Z_1, \ldots, Z_m\}$ una colección de cláusulas sobre $X = \{x_1, \ldots, x_n\}$, con tres literales cada una. Se construye un grafo G tal que G es hamiltoniano si, y sólo si, $\mathcal{Z}$ es satisfacible.



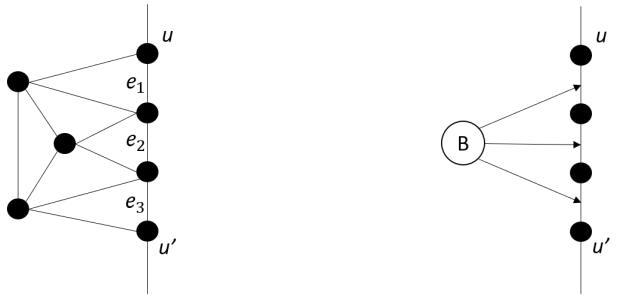
(a) (b)

Figura 1.3: Subgrafo A

En primer lugar, se definen dos subgrafos que aparecerán varias veces, a modo de patrón sobre G. Se considera por un lado el ilustrado en la Figura 1.3 (a), que denominaremos A, y se asume que ninguno de sus vértices exceptuando u, u', v, v' conectan con algún otro nodo de G. Entonces, cualquier ciclo hamiltoniano sobre G pasará por A en alguna de las formas mostradas en la Figura 1.4. Por tanto, es posible sustituir A por dos aristas, tal y como se ilustra en la Figura 1.3 (b), con la restricción adicional de que cualquier ciclo hamiltoniano sobre G debe contener exactamente una de ellas.

(a) (b)

Figura 1.4: Ciclos hamiltonianos posibles restringidos a A.



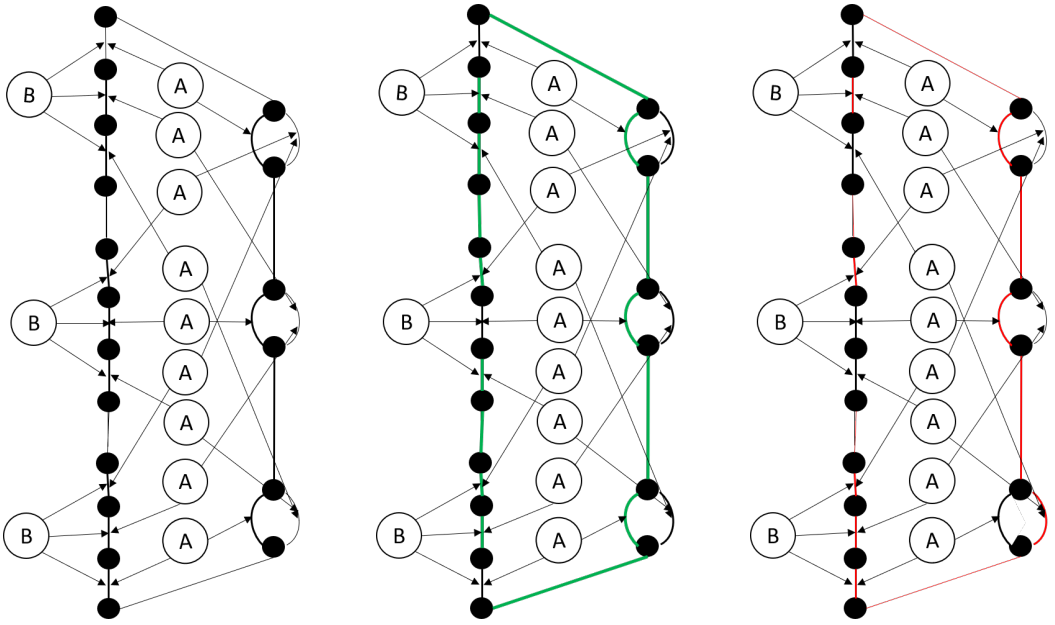
(a) (b)

Figura 1.5: Ciclos hamiltonianos posibles restringidos a A.

A continuación se considera el subgrafo B, ilustrado en la Figura 1.5 (a), y se asume que ninguno de sus vértices con excepción de u y u' conectan con cualquier otro nodo de G. Entonces ningún ciclo hamiltoniano sobre G atraviesa las aristas e_1, e_2 y e_3. Además, para cualquier $S \subsetneq \{e_1, e_2, e_3\}$ existe un camino hamiltoniano desde u hasta u' en B que contiene S pero ningún elemento de $\{e_1, e_2, e_3\} \setminus S$. Se representa B por la ilustración de la Figura 1.5 (b).

Con los pasos previos es posible construir G. Para cada cláusula se introduce una copia de B unidas secuencialmente. Entre la primera y la última copia de B se insertan dos vértices por cada variable unidos uno tras otro. Entonces se duplican las aristas entre los dos vértices de cada variable x; esas dos aristas se corresponderán con x y $\overline{x}$ respectivamente. Las aristas e_1, e_2, e_3 en cada copia de B quedan conectadas a través de una copia de A del primero, segundo y tercer literal de la correspondiente cláusula. Esta construcción se ilustra en la Figura 1.6.

Nótese que, las restricciones anteriores debidas a la construcción de A y de B imponen que en cada aparición de A, un ciclo hamiltoniano pasará por exactamente una de las dos formas que ilustra la Figura 1.4. A su vez, en cada aparición de B no existe ningún ciclo hamiltoniano que pueda atravesar las tres aristas e_1, e_2 y e_3 anteriores. De esta forma, siendo $X = \{x_1, x_2, x_3\}$, el circuito generado en la Figura 1.6 (b), representado en verde, con valor de verdad $T : L \to \{true, false\}$ (donde $L = X \cup \{\overline{x} : x \in X\}$) dado por $T(x) = true$ para todo $x \in X$ es hamiltoniano al satisfacer todas las restricciones impuestas por la construcción de los subgrafos A y B. Sin embargo, en la Figura 1.6 (c) sí se aprecia que el circuito generado, representado en rojo, incumple la restricción



(a) Grafo G

(b) Camino generado si $T(x) = true$ para todo $x \in \{x_1, x_2, x_3\}$

(c) Camino generado si $T(x) = true$ para todo $x \in \{x_1, x_2, \overline{x_3}\}$

Figura 1.6: Ejemplo de grafo G resultante de $\mathcal{Z} = \{\{x_1, \overline{x_2}, \overline{x_3}\}, \{\overline{x_1}, x_2, \overline{x_3}\}, \{\overline{x_1}, \overline{x_2}, x_3\}\}$.

que impone que ningún ciclo hamiltoniano puede pasar por las tres aristas que une un subgrafo B, no siendo por tanto hamiltoniano.

Por tanto, se prueba que G es hamiltoniano si, y sólo si $\mathcal{Z}$ es satisfacible.

Sea C un ciclo hamiltoniano, se define el valor de verdad T dado por

$$T(a) = \begin{cases} true & si \quad a \in C, \\ \\ false & si \quad a \notin C. \end{cases}$$

Por las propiedades de A y de B cada cláusula contendrá un literal cuyo valor es *true*, haciendo a $\mathcal{Z}$ satisfacible.

Recíprocamente, cualquier valor de verdad define un camino sobre el grafo G. Si T satisface $\mathcal{Z}$, puesto que cada cláusula contienen un literal de valor *true*, entonces el camino que se genera sobre G es un ciclo hamiltoniano. □

Con la construcción teórica previa, es posible concluir con la complejidad ya mencionada del problema del viajante TSP, resultado con el que ponemos fin a esta sección.

Corolario 1.38. *El problema TSP es fuertemente $\mathcal{NP}$-hard.*

Demostración. Dada la restricción al TSP que considera aquellas instancias donde todas las distancias entre ciudades son 1 o 2, veamos que el problema de los ciclos hamiltonianos reduce en la restricción anterior.

Dado un grafo $G = (V, E)$ con $V = \{v_1, v_2, \ldots, v_n\}$, se considera la instancia del TSP $\mathcal{I} = (G, C)$ donde $C = (c_{ij})_{1 \leq i,j, \leq n}$ viene dada por

$$c_i j = \begin{cases} true & si \quad (v_i, v_j) \in E, \\ \\ false & si \quad (v_i, v_j) \notin E. \end{cases} \tag{1.3.8}$$

En estas condiciones, es claro que G será hamiltoniano si, y sólo si, la solución óptima del problema TSP tiene longitud n. De esta forma queda visto que el problema de los ciclos hamiltonianos reduce polinomialmente en la restricción del TSP considerada, y por ser el primero $\mathcal{NP}$-*completo*, todos los problemas de clase $\mathcal{NP}$ se transforman en él, por lo que la restricción del TSP considerada es $\mathcal{NP}$-*hard*. □

1.4. Aplicaciones generales del TSP

El TSP es un problema de interés científico debido esencialmente a la complejidad computacional que requiere su resolución. Pero además, es ampliamente estudiado por sus múltiples utilidades prácticas.

En realidad, muchos supuestos pueden ser modelados a través del TSP (i.e. planteados como instancias del problema de cuya solución se deriva la del supuesto modelado). Así, existen muchas aplicaciones prácticas del problema del viajante, una de las cuales es la reordenación de los mensajes ADS-B asociados a una trayectoria, que motiva este Trabajo Fin de Grado y que se expone en el Capítulo 4.

Las aplicaciones más claras del TSP, tal y como fue concebido, están ligadas al enrutamiento de vehículos, donde se dispone de un conjunto de ubicaciones que deben ser visitadas por un determinado vehículo recorriendo la menor distancia posible. Pensemos por ejemplo en una compañía con servicio de venta por Internet que debe entregar una serie de paquetes en un conjunto de direcciones prefijadas a través de uno de sus vehículos.

Existen otras aplicaciones del problema del viajante entre las que se encuentran las siguientes [Laporte, 1992]:

1. El problema del *cableado de ordenador*. Algunos sistemas informáticos pueden describirse en términos de módulos que se unen entre sí por medio de cables. Se desea unir todos los módulos (ciudades) por medio de cables (aristas) de mínima longitud y de tal manera que cada módulo quede unido a otros dos.
2. El problema del *papel de pared*. Se quieren cortar n hojas de un rollo de papel tapiz en el cual se repite un patrón de longitud 1. Para cada hoja i, se denota a_i y b_i como los puntos del patrón donde comienza y finaliza la hoja. Entonces, si se corta la hoja j justo después que la hoja i se obtiene un desperdicio c_{ij} dado por

$$c_{ij} = \begin{cases} a_j - b_i & si \quad b_i \leq a_j, \\ \\ 1 + a_j - b_i & si \quad b_i > a_j. \end{cases}$$

Se pretende por tanto obtener las n hojas del rollo con el menor desperdicio posible. Para definir este problema en términos del TSP se consideran las hojas como ciudades y el gasto de papel de cortar la hoja j justo después que la hoja i como c_{ij} (longitud del camino que une la ciudad i con la ciudad j). A mayores, sería necesario introducir una hoja artificial $n+1$ a modo de unión entre la primera y la última hoja cortadas tal que $c_{i,j} = 0$ si $i = n+1$ ó $j = n+1$. De esta forma se tendrá que acabar en la ciudad de partida.

3. El problema de la *secuenciación de trabajos*. Se supone que una máquina debe realizar n trabajos de forma secuencial y que c_{ij} es el tiempo que la máquina invertiría en pasar del trabajo i al trabajo j. Entonces de nuevo, para minimizar el tiempo invertido en cambiar de un trabajo a otro (o *change-over time*) es posible formular el problema en términos del TSP introduciendo un trabajo artificial de coste 0, de forma análoga al problema anterior.

4. En *cristalografía*, algunos experimentos consisten en tomar un gran número de medidas sobre cristales a través de un detector. Cada muestra debe montarse en un aparato que debe ser posicionado para poder hacer la medición, invirtiendo tiempo en cambiar la posición del detector desde la que disponía para realizar la última medición hasta la requerida para realizar la actual. Como en los anteriores, añadiendo una ciudad artificial de coste 0 que una la primera muestra con la última obtenemos una instancia del TSP donde las ciudades son las muestras a examinar y el coste de las aristas que las unen es el tiempo invertido por el detector para cambiar su posición.

Capítulo 2

Algoritmos de resolución del TSP: las heurísticas de mejora local

En el Capítulo 1, se ha visto que el problema del viajante (TSP) es un problema de optimización combinatoria fuertemente $\mathcal{NP}$-*hard*. Esto implica que, a menos que se cumpla la igualdad $\mathcal{P} = \mathcal{NP}$ (a día de hoy sin demostrar), no existe ningún algoritmo de complejidad polinomial que garantice la solución óptima del problema.

En la actualidad, son varias las líneas de investigación establecidas a fin de desarrollar un algoritmo con estas propiedades, o al menos mejorar los ya existentes. El presente trabajo se centra en una de las corrientes más avanzadas en términos de efectividad de los métodos que abarca a la hora de abordar el problema del viajante: las *heurísticas de mejora local*. En concreto, en este capítulo se exponen las bases que guían este tipo de procedimientos así como sus algoritmos principales.

2.1. Principales algoritmos de resolución

Tal y como se adelanta en la Sección 1.2, los algoritmos que tratan de resolver el TSP se clasifican, según garanticen o no encontrar la solución óptima del problema al ser ejecutados, en *algoritmos exactos* y *algoritmos de aproximación*.

2.1.1. Los algoritmos exactos

Los *algoritmos exactos* son aquellos que buscan obtener la solución óptima del problema independientemente del coste computacional requerido. Estos algoritmos aseguran la obtención de la solución óptima, lo que implica un alto coste computacional ligado a las propiedades del TSP. Es por ello que su uso es preferible sobre instancias pequeñas, cuando el tiempo de ejecución de estos algoritmos aun no es desmedido.

Dentro de estos algoritmos encontramos el rudimentario *ataque por fuerza bruta* que prueba todas las permutaciones circulares posibles sobre los nodos del grafo, quedándose

con aquella que minimiza la expresión (1.1.1). La complejidad computacional de este algoritmo es por tanto $\mathcal{O}(n!)$ siendo n el número de ciudades de la instancia, asumible sobre instancias pequeñas, pero impensable sobre otras más grandes (con $n = 20$ este número es ya del orden de $2 \cdot 10^{18}$).

Una de las familias de algoritmos exactos más exitosas a lo largo de la historia del problema es la de los métodos *branch and bound*, o de *ramificación y acotación*, que consisten en dividir la instancia en otras más pequeñas, calculando cotas de la longitud de la ruta más corta de cada parte, y que se obtienen de "relajar" las hipótesis del problema. Tras obtener una solución para cada subinstancia, el algoritmo obtiene la ruta de la instancia principal, que resulta ser una solución óptima.

Los algoritmos *branch and bound* se basan en programación lineal de enteros, que parte de una formulación del problema a resolver. Para el TSP, Dantzig, Fulkerson y Johnson desarrollaron en 1954 una de las primeras [Lawler et al., 1992], y que consiste en que dada una instancia $\mathcal{I} = (G, C)$ donde G es un grafo de n vértices y $C = (c_{ij})_{1 \leq i,j \leq n}$ es la matriz de costes, se quiere minimizar la expresión

$$\sum_{i \neq j} c_{ij} x_{ij} \quad \text{con } x_{ij} \in \mathbb{Z}, \quad \text{para todo } i, j \text{ en } 1 \leq i, j \leq n \tag{2.1.1}$$

sujeto a

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, ..., n; \tag{2.1.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, ..., n; \tag{2.1.3}$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1; \tag{2.1.4}$$

$$S \subset V, \qquad 2 \leq |S| \leq n - 2; \tag{2.1.5}$$

$$x_{ij} \in \{0, 1\}, \qquad i, j = 1, \ldots, n, \ i \neq j. \tag{2.1.6}$$

Es necesario también citar el algoritmo de *Held-Karp*, que aparece en 1962 de la mano de Michael Held y Richard Karp, y que se basa en programación dinámica [Cook, 2012]. Capaz de encontrar la solución óptima de cualquier instancia del TSP, este algoritmo fija una ciudad como punto de partida, a partir de la cual va construyendo caminos formados por cada vez más vértices y de los cuales va almacenando su longitud. De esta forma, dado un vértice de destino, y un subconjunto de vértices intermedios, el algoritmo halla cual es la forma más corta de recorrer todos los nodos intermedios partiendo desde el primero fijado al comienzo y el último señalado. El método itera hasta abarcar todos los vértices. A pesar de obtener la solución óptima de cualquier instancia, el coste computacional de este algoritmo es $\mathcal{O}(n^2 2^n)$ lo que lo inhabilita para ser utilizado sobre instancias grandes.

En la actualidad, *Concorde* es uno de los algoritmos más destacados. Implementado por David Applegate, Robert Bixby, Vasek Chavátal y William Cook, consiste en uno de los desarrollos más recientes del *método de ramificación y acotación* sobre árboles de

búsqueda. Este algoritmo, cuyo código está disponible en la red, fue utilizado para obtener la solución óptima a instancias del problema de tamaño *record* hasta la fecha. Entre ellas se encuentran supuestos de: 3.038 ciudades en 1992, 13.509 en 1998, 24.978 en 2004 y 85.900 en 2006 Cook [2012].

2.1.2. Los algoritmos de aproximación

Por otro lado se encuentran los *algoritmos de aproximación*, que son aquellos que dan prioridad a un tiempo de ejecución razonable frente a encontrar la solución óptima.

Mientras que ejecutar un algoritmo exacto para el problema del viajante puede requerir un coste computacional inasumible para instancias de gran tamaño, un algoritmo de aproximación, aunque no encuentre la solución óptima, es mucho menos complejo computacionalmente, por lo que es posible ejecutarlos sobre instancias de mayor tamaño.

Los más sencillos de estos métodos son las *heurísticas constructivas*, que engloban una serie de métodos cuya mecánica consiste en construir una ruta partiendo de un nodo al que van uniendo sucesivamente ciudades en cada iteración del algoritmo. Algunos de los más destacados son los siguientes [Gutin y Punnen, 2007]:

- **Método de vecinos más próximos** (o *nearest neighbour*): es el método constructivo más simple. Dada una instancia del TSP $\mathcal{I} = (G, C)$ con $G = (V, E)$, $V = \{v_1, v_2, \ldots, v_n\}$ y $C = (c_{ij})_{1 \leq i,j \leq n}$ tal que c_{ij} es el coste asociado a la arista (v_i, v_j) para cada $1 \leq i, j \leq n$. Se toma un vértice, que supongamos sin pérdida de generalidad que es $v_1 \in V$. En cada iteración $k = 2, 3, \ldots, n$, se tiene un camino, que supongamos sin pérdida de generalidad que es $P = (v_1, v_2, \ldots, v_k)$, se construye un nuevo camino $P' = (v_1, v_2, \ldots, v_k, w)$ tal que $w \in V \setminus \{v_1, v_2, \ldots, v_k\}$ verifica que

$$c_k^* = \min_{k+1 \leq x \leq n} c_{kx}$$

  siendo c_k^* el coste de la arista (v_k, w). Tras las n iteraciones se obtiene un camino hamiltoniano que a continuación se cierra añadiendo el primer nodo al final.

- **Método de vecinos más próximos iterativo** (o *repetitive nearest neightbor*): variante algo más compleja del método de vecinos más próximos. Dada la instancia del TSP $\mathcal{I} = (G, C)$ con $G = (V, E)$, $V = \{v_1, v_2, \ldots, v_n\}$ y sea $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ el conjunto de rutas sobre G tal que R_i es la solución obtenida por el método de *vecinos más próximos* cuando se toma v_i como vértice de partida. Entonces, el método de *vecinos más próximos iterativo* devuelve la ruta $R \in \mathcal{R}$ de mínima longitud.

- **Greedy** (*codicioso*): dada una instancia del TSP $\mathcal{I} = (G, C)$ con $G = (V, E)$ donde $V = \{v_1, v_2, ..., v_n\}$, y sea $C = (C_{ij})_{1 \leq i,j \leq n}$ tal que c_{ij} es el coste de la arista $(v_i, v_j) \in E$. Dado $E = \{e_1, e_2, ..., e_m\}$ donde si $x < y$ se tiene que, siendo $e_x = (v_{a_1}, v_{a_2})$ y $e_y = (v_{b_1}, v_{b_2})$, $c_{a_1,a_2} < c_{b_1,b_2}$. Se establece la relación de equivalencia $\Re$ sobre el conjunto de aristas E, dada por

$$(v_{a_1}, v_{a_2}) \Re (v_{b_1}, v_{b_2}) \Leftrightarrow \exists \alpha, \beta \in \{1, 2\} \text{ tales que } v_{a_\alpha} = v_{b_\beta}$$

Se construye un conjunto Γ de n aristas de la forma siguiente. Para cada $k = 1, 2, \ldots, m$, se añade e_k a Γ si

$$\nexists e \in \Gamma \text{ tal que } e_k \Re e$$

En este caso, si al añadir e_k a Γ se tiene que $|\Gamma| = n$, se deriva la ruta R sobre G tal que $E(R) = \Gamma$ y se detiene la ejecución. Su complejidad es $\mathcal{O}(n^2 log(n))$.

- **Inserción más barata** (o *cheapest insertion*): dada la instancia del TSP $\mathcal{I} = (G, C)$ con $G = (V, E)$ con $V = (v_1, v_2, \ldots, v_n)$, y $C = (c_{ij})_{1 \leq i,j \leq n}$ tal que c_{ij} es el coste asociado a la arista $(v_i, v_j) \in E$. Se parte de un nodo y su vecino más próximo, que supongamos sin pérdida de generalidad que son $v_1, v_2 \in V$ respectivamente. Para cada iteración $k = 2, 3, \ldots, n-1$, se tiene, suponiendo de nuevo sin pérdida de generalidad, el camino $P = (v_1, v_2, \ldots, v_k)$, se obtiene el camino $P' = (v_1, v_2, \ldots, v_{l_1}, v_{l_2}, v_{l_1+1}, \ldots, v_n)$ tal que $1 \leq l_1, l_2 \leq n, l_1 \neq l_2$ verifican que

  $$c_{l_1,l_2} + c_{l_2,l_1+1} - c_{l_1,l_1+1} = \text{mín}\{c_{\alpha,\beta} + c_{\beta,\alpha+1} - c_{\alpha,\alpha+1} : k \leq \alpha, \beta \leq n,\ \beta \notin \{\alpha, \alpha+1\}\}$$

  Se concluye cerrando el camino hamiltoniano obtenido.

- **Inserción más cercana** (o *nearest neightbor*): dada la instancia del TSP $\mathcal{I} = (G, C)$ con $G = (V, E)$ con $V = (v_1, v_2, \ldots, v_n)$, y $C = (c_{ij})_{1 \leq i,j \leq n}$ tal que c_{ij} es el coste asociado a la arista $(v_i, v_j) \in E$. Se inicia el algoritmo con una ciudad de partida y su vecino más cercano que supongamos, sin pérdida de generalidad, que son $v_1, v_2 \in V$ respectivamente. Para cada iteración $k = 3, 4, \ldots, n$, se supone de nuevo sin pérdida de generalidad que se tiene el camino $P = (v_1, v_2, \ldots, v_{k-1})$. Se añade la arista $v_\alpha \in V \setminus \{v_1, v_2, \ldots, v_{k-1}\}$ tal que existe $\beta \in \mathbb{N}$ con $1 \leq \beta \leq (k-1)$ verificando

  $$c_{\alpha,\beta} = \text{mín}\{c_{i,j} : 1 \leq j < k \leq i \leq n\}.$$

- **Inserción más lejana** (o *farthest neightbor*): dada la instancia del TSP $\mathcal{I} = (G, C)$ con $G = (V, E)$ con $V = (v_1, v_2, \ldots, v_n)$, y $C = (c_{ij})_{1 \leq i,j \leq n}$ tal que c_{ij} es el coste asociado a la arista $(v_i, v_j) \in E$. Se inicia el algoritmo con una ciudad de partida y su vecino más cercano que supongamos, sin pérdida de generalidad, que son $v_1, v_2 \in V$ respectivamente. Para cada iteración $k = 3, 4, \ldots, n$, se supone sin pérdida de generalidad el camino $P = (v_1, v_2, \ldots, v_{k-1})$ y se denota por $V_k = \{v_1, v_2, \ldots, v_{k-1}\}$ y por $W_k = V \setminus V_k$. Se añade la arista $v_\alpha \in W_k$ tal que existe $\beta \in \mathbb{N}$ con $1 \leq \beta \leq (k-1)$ verificando

  $$c_{\alpha,\beta} = \max_{v_i \in W_k} \min_{v_j \in V_k} c_{i,j}.$$

- **Inserción arbitraria** (o *arbitrary insertion*): muy similar al algoritmo de *inserción más lejana*, pero difiere del anterior en que a cada iteración se añade a la ruta en construcción un nodo aleatorio de los restantes.

Una de las familias de algoritmos de aproximación más importantes son los *búsqueda local*, también llamados *heurísticas de mejora local*, cuyo análisis es objeto de este Trabajo Fin de Grado y que se profundizan en la Sección 2.2). Estos métodos parten de una ruta inicial (solución factible) que van modificando a cada paso para obtener otras rutas más

cortas. Se caractericen por la técnica de modificación que utilicen, que restringe la cantidad de rutas que es posible obtener desde una dada.

Los más conocidos son los algoritmos de la familia *k-Opt*, que en cada iteración buscan una nueva solución reemplazando k aristas de la ruta actual por otras k nuevas mediante un procedimiento de modificación que recibe el nombre de *k-change* (ver Definición 2.6). Los métodos más conocidos de esta familia son el *2-Opt* y el *3-Opt*. Por otro lado, encontramos una implementación más sofisticada de los algoritmos *k-Opt* en el *algoritmo de Lin-Kernighan*, que a diferencia de los anteriores, varía el valor de la k a lo largo de su ejecución con el fin de adaptarse al tipo de modificaciones que resultarán más efectivas a cada instante. Por otro lado, encontramos el algoritmo *Or-Opt* que basa su procedimiento en la escisión de cadenas de vértices de la ruta actual para unirlas en otra posición.

Otros algoritmos de aproximación con aplicación al TSP son los siguientes:

- *Simulated annealing*: en 1983, se introduce a través de uno de los trabajos de Kirkpatrick, Gelatt y Vecchi el *algoritmo de enfriamiento simulado* o *simulated annealing* como método de resolución del problema TSP con unos resultados bastante prometedores. Mientras que un algoritmo típico de búsqueda local sigue la estrategia *hill-climbing*, consistente en realizar solo movimientos hacia soluciones vecinas mejores que la actual, el *simulated annealing* realiza saltos arbitrarios sobre el espacio de estados con una cierta frecuencia a fin de no quedar atrapado en máximos locales. La frecuencia y la distancia de estos saltos va disminuyendo con el paso de las iteraciones (se produce el *enfriamiento*).
- Los *algoritmos de redes neuronales*: basados en el funcionamiento del cerebro humano. Sus componentes básicos son unidades de cómputo simples que reciben el nombre de *neuronas artificiales* que se comunican entre si a través de una red de conexiones que simulan el procesamiento en paralelo que ocurre en el cerebro humano. Este modelo fue aplicado al TSP por Hopfield y Tank en 1985.
- Los *algoritmos genéticos*: trabajan con un conjunto de soluciones del problema. En cada iteración, se eligen las mejores soluciones (*selección*), las cuales se combinan para obtener otras nuevas (*cruce*), finalmente se alteran (*mutación*) y se repite el proceso tomando estas nuevas soluciones como entrada de la siguiente iteración.

2.2. Las heurísticas de mejora local

A lo largo de esta sección se describen algunos de los algoritmos de búsqueda local más destacados para la resolución del problema TSP euclídeo. Todos estos métodos comparten el hecho de partir de una ruta sobre la que se van aplicando sucesivas modificaciones alcanzando otras soluciones factibles de menor longitud. En función de las modificaciones que es posible realizar, se define el denominado *grafo de estados* de una instancia.

Definición 2.1. *Dada una instancia $\mathcal{I} = (G, C)$ del problema TSP y un algoritmo de búsqueda local, se define el* ***grafo de estados*** *asociado a $\mathcal{I}$ como aquel en el que los*

*nodos son cada uno de los **estados** o rutas que se pueden formar sobre G. Además dos estados quedan unidos por una arista si es posible, a través de una iteración del algoritmo, transformar el estado de partida en el de llegada.*

Según las técnicas de modificación que utilicen, los algoritmos de búsqueda local dividen en:

- **Algoritmos de intercambio de aristas (EE: *edge exchanges*)**: se basan en la sustitución de un subconjunto de aristas de la ruta que son reemplazadas por otras nuevas. Los más conocidos son los algoritmos *k-Opt* (ver Subsección 2.2.1 y Subsección 2.2.2), así como el de *Lin-Kernighan*.
- **Algoritmos de intercambio de cadenas (CE: *chain exchanges*)**: en cada iteración, consideran una cadena de vértices de la ruta, que son desligados del resto para volver a unirlos en otra posición mediante la eliminación de una arista. En la Sección 2.2.4 se introduce el algoritmo *Or-Opt*, que sigue esta mecánica.

2.2.1. Análisis de la complejidad del algoritmo 2-Opt

El algoritmo *2-Opt* es el más simple de las heurísticas de la familia *k-Opt*. Las modificaciones que realiza sobre la ruta actual reciben el nombre de *2-changes*.

Definición 2.2. *Dada una instancia $\mathcal{I} = (G, C)$ donde $G = (V, E)$, y una ruta R sobre G, una modificación sobre R que da como resultado otra ruta R' sobre G se dice que es de tipo **2-change** si R y R' difieren únicamente en que se han eliminado dos aristas $(u_1, u_2), (v_1, v_2) \in E(R)$ reemplazadas por $(u_1, v_1), (u_2, v_2) \in E(R')$.*

Algoritmo 1 2-Opt

Entrada: Una instancia del TSP $\mathcal{I} = (G, C)$, con $G = (V, E)$ y $C = (c_{i,j})_{i,j=1}^{n}$.
Salida: Una ruta R

```
Sea R una ruta
Sea S la familia de subconjuntos de E(R) con dos elementos
para S ∈ S y toda ruta R' con E(R') ⊇ E(R) \ S hacer
  si Σ_{(v_i,v_j)∈E(R)'} c_{i,j} < Σ_{(v_i,v_j)∈E(R)} c_{i,j} entonces
    Asignar R := R'
    Sea S la familia de subconjuntos de E(R) con dos elementos
  fin si
fin para
devolver R
```

El Algoritmo 1 presenta el pseudocódigo asociado al método *2-Opt* [Korte y Vygen, 2006]. En cada iteración, se prueba un *2-change*, que en caso de mejorar la ruta actual R, la sustituye por la nueva obtenida, y vuelve a construir el conjunto $\mathcal{S}$ de vecinos de la ruta actual en la matriz de estados, sobre el que itera el bucle.

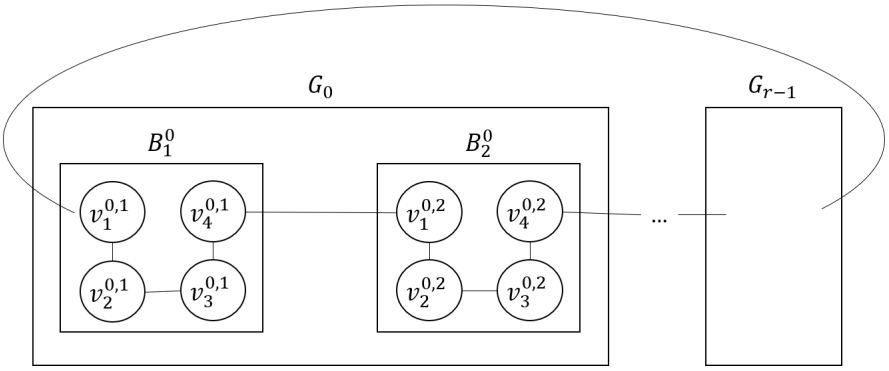


Figura 2.1: Modelo de la construcción para el análisis de complejidad del algoritmo *2-Opt* [Englert et al., 2007]

El método *2-Opt* alcanza, por lo general, un orden de complejidad subcuadrático [Johnson y McGeoh, 1997], si bien existen casos en los que el *2-Opt* alcanza orden exponencial. En esta sección se prueba este hecho mediante la construcción de instancias sobre las que puede llegar a darse este orden de complejidad. Seguiremos para ello el enfoque dado en Englert et al. [2007].

Sea $\mathcal{I} = (G, C)$ una instancia del TSP con $G = (V, E)$ y R una ruta sobre G. Supongamos que V admite una partición $P = \{G_0, G_1, \ldots, G_{r-1}\}$ donde cada G_i recibe el nombre de *gadget* (o *artefacto*) y contiene 8 vértices divididos en 2 componentes denominados *bloques*, cada uno de ellos con cuatro vértices.

Dado el *gadget* G_i, se denotan sus bloques por B_1^i y B_2^i y por $v_1^{i,j}, v_2^{i,j}, v_3^{i,j}$ y $v_4^{i,j}$ los cuatro vértices (o ciudades) que componen el bloque B_j^i $(j = 1, 2)$.

Se supone que la ruta R visita los cuatro vértices de cada bloque formado consecutivamente y que $v_1^{i,j}$ y $v_4^{i,j}$ son la primera y última ciudad del bloque respectivamente en ser visitadas para cada i en $1 \leq i \leq r-1$ y $j \in \{1, 2\}$. Así, solo se pueden dar las configuraciones $v_1^{i,j} v_2^{i,j} v_3^{i,j} v_4^{i,j}$ y $v_1^{i,j} v_3^{i,j} v_2^{i,j} v_4^{i,j}$.

Suponiendo además que

$$d(v_1^{i,j}, v_2^{i,j}) + d(v_3^{i,j}, v_4^{i,j}) < d(v_1^{i,j}, v_3^{i,j}) + d(v_2^{i,j}, v_4^{i,j}).$$

se tiene que la configuración $v_1^{i,j} v_2^{i,j} v_3^{i,j} v_4^{i,j}$ es más corta que $v_1^{i,j} v_3^{i,j} v_2^{i,j} v_4^{i,j}$ denotándolas por *estado S (short)* y *estado L (long)* del bloque B_j^i respectivamente. Así, se representan por $(L, S), (S, L), (L, L)$ y (S, S) las posibles combinaciones de estados de los bloques de un *gadget* donde la primera y segunda componente se refieren al estado del bloque B_1^i y B_2^i respectivamente. Las dos últimas combinaciones reciben el nombre de *estado inicial* y *estado final* del *gadget* respectivamente.

La siguiente propiedad es verificada por las instancias que responden a la construcción anterior, y es la base para la prueba de los resultados posteriores de la sección.

Propiedad 2.3. *Si G_i con $1 \leq i \leq r-2$ está en su estado inicial ((S, L) respectivamente) y G_{i+1} está en su estado final, entonces existe una secuencia de siete 2-changes que*

conducen a G_i al estado (S, L) ((S, S) respectivamente) y a G_{i+1} al estado inicial.

Demostración. Supongamos que el *gadget* G_i está en estado (L, L) y el *gadget* G_{i+1} está en su estado inicial. Entonces, la secuencia de *2-changes* representada en la Figura 2.2 concluiría con G_i en su estado (S, L) y G_{i+1} en su estado inicial.

La Figura 2.3 muestra la secuencia de siete *2-changes* que transforma los *gadgets* G_i y G_{i+1} de su estado (S, L) a (S, S), y (S, S) a (L, L) respectivamente. □

La satisfacción de la propiedad anterior garantiza la existencia de una secuencia de *2-changes* de complejidad exponencial.

Lema 2.4. *Sea G un grafo y $P = \{G_0, G_1, ..., G_{r-1}\}$ su partición de gadgets, tal que G_i está en su estado (L, L) y todo gadget G_j, con $j > i$ está en su estado (S, S). Entonces, existe una secuencia de $2^{r+3-i} - 14$ 2-changes que concluye con todo G_j tal que $j \geq i$ en su estado (S, S).*

Demostración. El lema se prueba por inducción sobre i.

- Si G_{r-1} se encuentra en su estado (L, L), puede pasar al estado (S, S) con dos *2-changes*: el que hace que pase de (L, L) a (S, L) y el que lo hace de (S, L) a (S, S), verificándose así que $2 = 2^{r+3-(r-1)} - 14$.
- Se supone que el lema se verifica para $i+1$ y se prueba para i. Si el *gadget* G_i está en su estado (L, L) y para todo G_j con $j > i$ están en estado (S, S), por la Propiedad 2.3 existe una secuencia de siete *2-changes* que concluye con G_i en estado (S, L) y G_{i+1} en estado (L, L). Por hipótesis de inducción, aplicando otros $2^{r+3-(i+1)} - 14$ *2-changes*, todo *gadget* G_j con $j > i$ estará en su estado (L, L). De nuevo por la Propiedad 2.3, existe una secuencia de siete *2-changes* que concluye con G_i en estado (S, S) y G_{i+1} en estado (L, L). Por hipótesis de inducción, aplicando otros $2^{r+3-(i+1)} - 14$ *2-changes*, se concluye que existe la secuencia de *2-changes* pedida con longitud
$$7 + 7 + 2 \cdot 2^{r+3-(i+1)} - 14 = 2^{r+3-i} - 14.$$

□

Con todo lo expuesto, Englert et al. [2007] concluye con el siguiente resultado, prueba de la existencia de esa secuencia de mejoras de tipo *2-change* de longitud exponencial en instancias del TSP euclídeo en función del tamaño de la misma.

Teorema 2.5. *Para todo $r \in \mathbb{N}$ existe un grafo en el plano euclídeo con $8n$ vértices cuyo grafo de estados correspondiente contiene una ruta de longitud $2^{r+3} - 14$.*

Demostración. Se considera una instancia del problema TSP euclídeo sobre el plano $\mathbb{R}^2$, siendo G el grafo de la instancia y G_i, con $0 \leq i \leq r-1$ los *gadgets* del grafo, donde G_0 se encuentra en su estado inicial y G_j, para todo $i < j \leq r-1$ se encuentra en su estado final. Entonces, en virtud del Lema 2.4 existe una secuencia de $2^{r+3} - 14$ *2-changes*, donde G tiene $8r$ vértices al estar formado, por definición, cada *gadget* por 8.

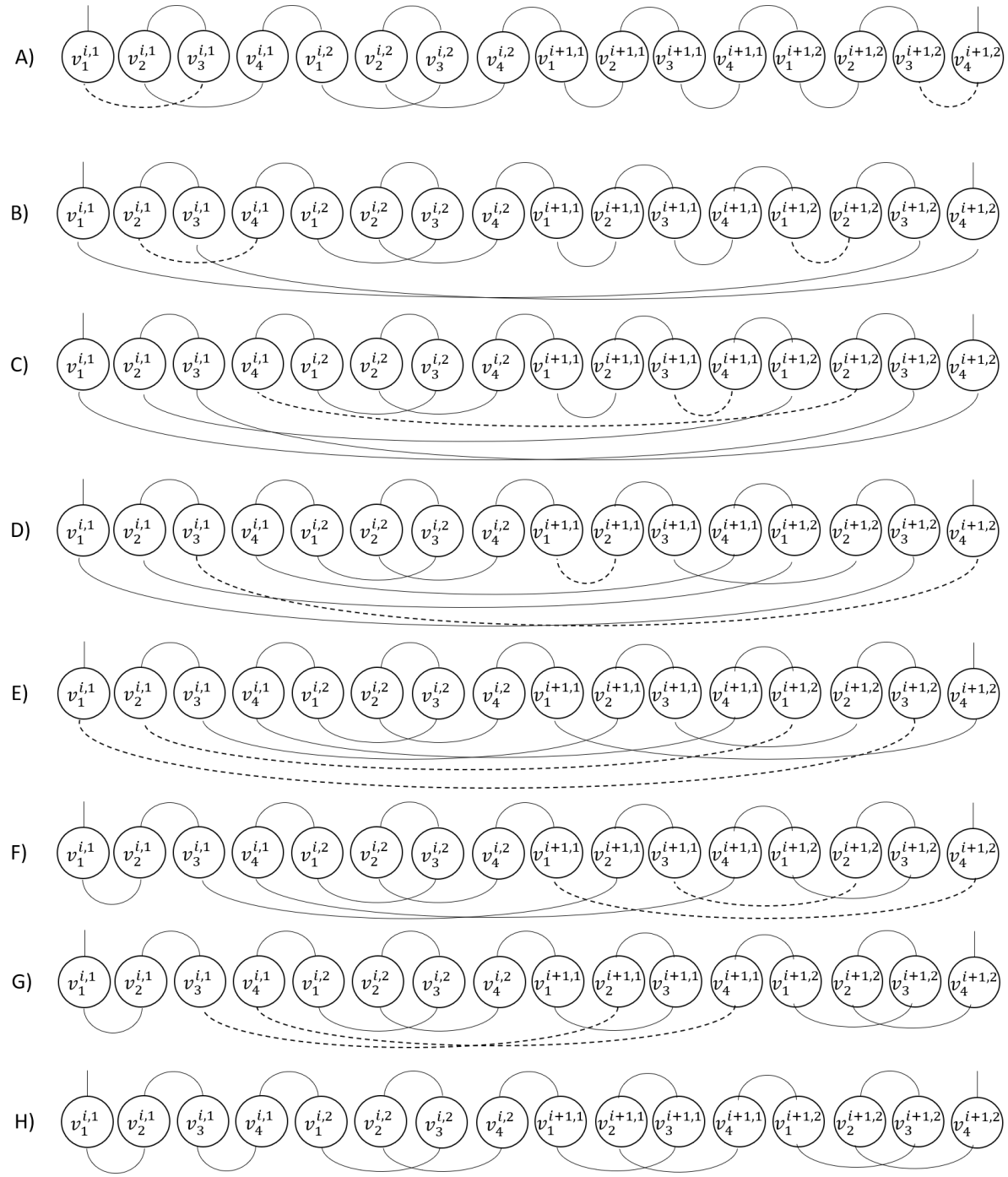


Figura 2.2: Estados por los que pasa la secuencia de nodos de los bloques B_1^i B_2^i del *gadget* G_i y B_1^{i+1} y B_2^{i+1} del *gadget* G_{i+1} para conducir al *gadget* G_i desde su estado (L, L) hasta su estado (S, L) y al *gadget* G_{i+1} desde su estado (S, S) hasta su estado (L, L)

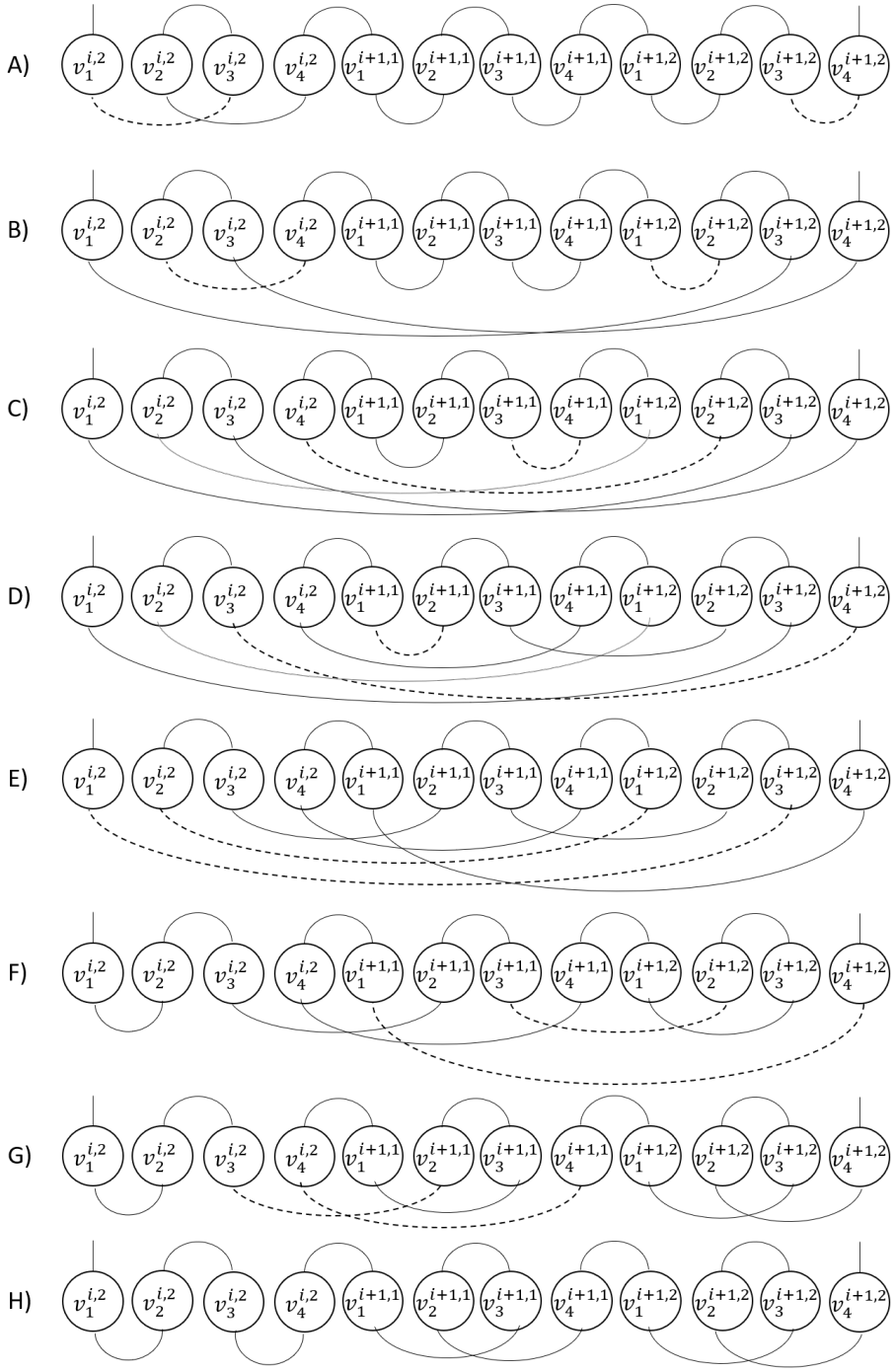


Figura 2.3: Estados por los que pasa la secuencia de nodos de los bloques B_2^i del *gadget* G_i y B_1^{i+1} y B_2^{i+1} del *gadget* G_{i+1} para conducir al *gadget* G_i desde su estado (S, L) hasta su estado (S, S) y al *gadget* G_{i+1} desde su estado (S, S) hasta su estado (L, L)

Por tanto, solo restaría encontrar la disposición de los puntos sobre el plano euclídeo tal que todos y cada uno de los cambios de tipo *2-change* que componen el camino anterior son a su vez mejoras (i.e. que la nueva ruta que se genera a cada paso es de menor longitud que la se obtuvo en el paso anterior). Para ello se especifica la posición de cada punto del último *gadget* y se da una regla que describe como puede derivarse la posición de los puntos de un *gadget* a partir de la posición de los puntos del *gadget* siguiente.

Sean $v_1^{r-1,k} = (0,0), v_2^{r-1,k} = (1,0), v_3^{r-1,k} = (-0.1, 1.4)$ y $v_4^{r-1,k} = (-1.1, 4.8)$ los nodos del bloque B_k^{r-1} para cada $k \in \{1,2\}$. Es claro que $v_1^{r-1,k} v_2^{r-1,1} v_3^{r-1,1} v_4^{r-1,1}$ se corresponde con el estado corto de B_k^{r-1} mientras que $v_1^{r-1,1} v_3^{r-1,1} v_2^{r-1,1} v_4^{r-1,1}$ lo hace con su estado largo, basta ver que

$$d(v_1^{r-1,k}, v_3^{r-1,k}) + d(v_2^{r-1,k}, v_4^{r-1,k}) > d(v_1^{r-1,k}, v_2^{r-1,k}) + d(v_3^{r-1,k}, v_4^{r-1,k}),$$

cierto, ya que

$$d(v_1^{r-1,k}, v_3^{r-1,k}) + d(v_2^{r-1,k}, v_4^{r-1,k}) = \sqrt{0.1^2 + 1.4^2} + \sqrt{2.1^2 + 4.8^2} > 6.64$$

y

$$d(v_1^{r-1,k}, v_2^{r-1,k}) + d(v_3^{r-1,k}, v_4^{r-1,k}) = \sqrt{1^2 + 0^2} + \sqrt{1^2 + 3.4^2} < 4.55$$

Ahora se establece la siguiente recurrencia para los *gadgets* anteriores:

1. Situar inicialmente los nodos de G_i en las posiciones de los nodos del *gadget* G_{i+1}.
2. Aplicar una rotación a los puntos del *gadget* G_i de $3\pi/2$ con centro en el origen.
3. Aplicar ahora sobre los puntos una homotecia de factor 3 y centro el origen.
4. Concluir trasladando los puntos por el vector $(-1.2, 0.1)$.

Así, para el *gadget* G_{r-2}, las posiciones de los nodos de $B_k^{r-2}, k \in \{1,2\}$ serían $v_1^{r-2,k} = (-1.2, 0.1), v_2^{r-2,k} = (-1.2, -2.9), v_3^{r-2,k} = (3, 0.4)$ y $v_4^{r-2,k} = (13.2, 3.4)$.

Puesto que en esta construcción cada *gadget* no es más que una copia del *gadget* posterior a la que se ha aplicado una rotación, una homotecia y una traslación, si para el par de *gadgets* G_{r-2} y G_{r-1} las secuencias de *2-changes* planteadas en la demostración del Lema 2.4 son a su vez mejoras (que acortan la nueva ruta obtenida), entonces también lo serán para cualquier par de *gadgets* G_i y G_{i+1} con $0 \leq i \leq r-2$.

Para la secuencia que parte con G_{r-2} en (L,L) y G_{r-1} en (S,S) y finaliza con G_{r-2} en (S,L) y G_{r-1} en (L,L), se obtienen las siguientes mejoras en longitud.

(1) $d(v_1^{r-2,1}, v_3^{r-2,1}) + d(v_3^{r-1,2}, v_4^{r-1,2}) - d(v_1^{r-2,1}, v_3^{r-1,2}) - d(v_3^{r-2,1}, v_4^{r-1,2}) > 0.03$

(2) $d(v_2^{r-1,2}, v_1^{r-1,2}) + d(v_4^{r-2,1}, v_2^{r-2,1}) - d(v_2^{r-1,2}, v_4^{r-2,1}) - d(v_1^{r-1,2}, v_2^{r-2,1}) > 0.91$

(3) $d(v_2^{r-1,2}, v_4^{r-2,1}) + d(v_3^{r-1,1}, v_4^{r-1,1}) - d(v_2^{r-1,2}, v_3^{r-1,1}) - d(v_4^{r-2,1}, v_4^{r-1,1}) > 0.05$

(4) $d(v_2^{r-1,1}, v_1^{r-1,1}) + d(v_3^{r-2,1}, v_4^{r-1,2}) - d(v_2^{r-1,1}, v_3^{r-2,1}) - d(v_1^{r-1,1}, v_4^{r-1,2}) > 0.04$

(5) $d(v_1^{r-2,1}, v_3^{r-1,2}) + d(v_2^{r-2,1}, v_1^{r-1,2}) - d(v_1^{r-2,1}, v_2^{r-2,1}) - d(v_3^{r-1,2}, v_1^{r-1,2}) > 0.43$

(6) $d(v_3^{r-1,1}, v_2^{r-1,2}) + d(v_1^{r-1,1}, v_4^{r-1,2}) - d(v_3^{r-1,1}, v_1^{r-1,1}) - d(v_2^{r-1,2}, v_4^{r-1,2}) > 0.06$

(7) $d(v_3^{r-2,1}, v_2^{r-1,1}) + d(v_4^{r-2,1}, v_4^{r-1,1}) - d(v_3^{r-2,1}, v_4^{r-2,1}) - d(v_2^{r-1,1}, v_4^{r-1,1}) > 0.53$

Nótese que, al haberse escogido las mismas coordenadas para los puntos del bloque $B_1^{r-1,1}$ y los de $B_2^{r-1,1}$, se obtienen los mismos datos de mejora para la secuencia de *2-changes* que comienza con G_{r-2} en su estado (S, L) y G_{r-1} en su estado (S, S), y que termina con G_{r-2} en su estado (S, S) y G_{r-1} en su estado (L, L). □

El autor indica la posibilidad de elegir diferentes posiciones para los nodos de los bloques B_1^{r-1} y B_2^{r-1} variando éstas ligeramente.

El estudio de Englert et al. [2007] concluye dejando como cuestiones abiertas la existencia o no de una instancia del problema en donde cualquier ejecución del algoritmo *2-Opt* describa siempre en su grafo de estados un camino de longitud exponencial, así como encontrar una mejor cota superior sobre el número de *2-changes* realizados por el algoritmo.

2.2.2. Algoritmos k-opt

La familia de los métodos *k-Opt* es la más conocida de los algoritmos de búsqueda local del tipo EE (*edge exchange*). Por su buen comportamiento, los más utilizados actualmente son el *2-Opt*, estudiado en la sección anterior, y el *3-Opt*.

Los algoritmos de la familia *k-Opt* se basan todos ellos en una mecánica similar a la hora de hacer las modificaciones sobre la ruta a partir de la que inician su ejecución. Estas modificaciones que realizan reciben el nombre de *k-change*.

Definición 2.6. *Dada una instancia $\mathcal{I} = (G, C)$ donde $G = (V, E)$ es un grafo de n vértices, y sea R una ruta sobre G. Una modificación de R que da como resultado otra ruta R' sobre G se dice que es de tipo* ***k-change*** *si a partir de R dado por*

$$R = (v_1^1, v_2^1, \ldots, v_{n_1}^1, v_1^2, v_2^2, \ldots, v_{n_2}^2, \ldots, v_1^k, v_2^k, \ldots, v_{n_k}^k, v_1^1),$$

se obtiene la ruta

$$R' = (v_1^1, v_2^1, \ldots, v_{n_1}^1, v_{n_2}^2, v_{n_2-1}^2, \ldots, v_1^2, v_{n_3}^3 \ldots, v_1^{k-1}, v_{n_k}^k, v_{n_k-1}^k, \ldots, v_1^k, v_1^1),$$

Nótese que la Definición 2.2 es una particularización de la Definición 2.6 con $k = 2$.

En el Algoritmo 2, se expone la implementación en pseudocódigo incluida en Korte y Vygen [2006] para la familia de métodos *k-Opt*. El algoritmo prueba un *k-change* sobre la ruta en cada iteración, y en caso de encontrar una ruta más corta, la asigna a R y reconstruye el conjunto de vecinos $\mathcal{S}$ para continuar iterando hasta que la ruta actual no admita más mejoras (i.e ninguna de las rutas vecinas sea mejor que la actual).

Algoritmo 2 Algoritmo k-Opt

Entrada: Una instancia del TSP $\mathcal{I} = (G, C)$, con $G = (V, E)$.
Salida: Una ruta R

1: Sea R una ruta
2: Sea $\mathcal{S}$ la familia de subconjuntos de $E(R)$ con k elementos
3: **para** $S \in \mathcal{S}$ y toda ruta R' con $E(R') \supseteq E(R) \setminus S$ **hacer**
4: **si** $\sum_{(v_i,v_j)\in E(R')} c_{i,j} < \sum_{(v_i,v_j)\in E(R)} c_{i,j}$ **entonces**
5: Asignar $R := R'$
6: Sea $\mathcal{S}$ la familia de subconjuntos de $E(R)$ con k elementos
7: **fin si**
8: **fin para**
9: **devolver** R

A pesar de la proximidad de las rutas obtenidas por estos algoritmos a la solución óptima, en el peor caso, la complejidad temporal de los mismos llega a ser exponencial, tal y como se prueba en Chandra et al. [1999].

El comportamiento del algoritmo vendrá determinado por el valor de k escogido. Mayores valores de k son recomendables cuando la ruta inicial dista mucho de la solución óptima, pues es cuando se requieren modificaciones más grandes sobre la ruta.

Sin embargo elegir el valor de k más adecuado no siempre es posible, pues en casos en que se desconoce el potencial de mejora de la primera solución. Es por ello que nace el *algoritmo de Lin-Kernighan*, el cual, basado en los algoritmos *k-Opt*, va variando el valor de k a lo largo de su ejecución para alcanzar mejores soluciones.

2.2.3. Algoritmo Lin-Kernighan

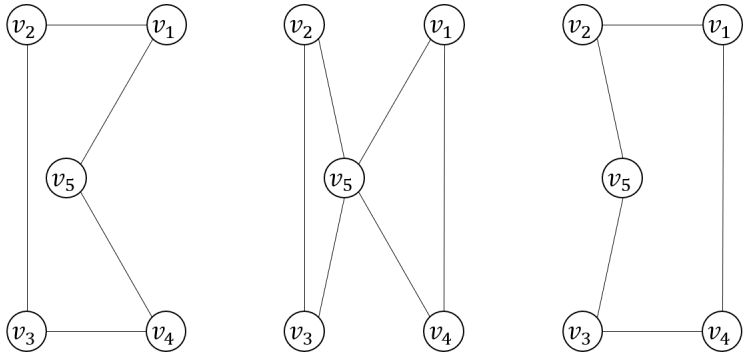
Para introducir los resultados teóricos que hay debajo del funcionamiento del *algoritmo de Lin-Kernighan* nos guiaremos del enfoque seguido en Korte y Vygen [2006], que centra su exposición en el concepto de *camino alterno*.

Definición 2.7. *Sea $\mathcal{I} = (G, C)$ una instancia del TSP, con $C = (c_{ij})_{i,j=1}^{n}$ y sea R una ruta sobre G. Un* ***camino alterno*** *$P = (v_1, v_2, \ldots, v_{2m+1})$ sobre G es aquel que verifica $(v_i, v_{i+1}) \neq (v_j, v_{j+1})$ para todo $1 \leq i < j < 2m + 1$, y $(v_i, v_{i+1}) \in R$ si, y sólo si i es par.*

La ***ganancia de P*** *viene dada por*

$$g(P) = \sum_{i=0}^{m-1} c_{2i+1,2i+2} - c_{2i+2,2i+3}$$

Se dice que P es ***adecuado*** *si $g(v_1, v_2, \ldots, v_{2i+1}) > 0$ para todo $i \in \{1, \ldots, m\}$.*



(a) Ruta R sobre el grafo

(b) Camino alterno P

(c) Nueva ruta con aristas $E(P) \vartriangle E(R)$

Figura 2.4: Ejemplo de camino alterno

La Figura 2.4 ilustra la principal aplicación de los caminos alternos al TSP. Dada una ruta (imagen (a)), es posible encontrar un camino alterno (imagen (b)), que combinado con la ruta, de como resultado otra ruta más corta que la anterior (imagen (c)). Esta idea, es la que lleva a la práctica el *algoritmo de Lin-Kernighan*.

La problemática reside entonces en encontrar para una ruta dada un camino alterno cuya combinación con la ruta (diferencia simétrica de los conjuntos de aristas) de como resultado una nueva ruta más corta que la anterior. El Lema 2.8 brinda una condición suficiente para que esto se de.

Notación. *Sea $\mathcal{I} = (G, C)$ una instancia del problema TSP, sea $G = (V, E)$ y $X \subset E$. Se denota por $Coste(X) = \sum_{(v_i,v_j)\in X} c_{i,j}$.*

Lema 2.8 (Lin & Kernighan, 1973)**.** *Sea R una ruta y P un camino alterno con $g(P) > 0$. Entonces:*

(a) $Coste(E(R) \vartriangle E(P)) = Coste(E(R)) - g(P)$,

(b) existe un camino alterno adecuado Q tal que $E(Q) = E(P)$.

Demostración. Para la parte (a), sea $P = (v_1, v_2, \ldots, v_{2m+1})$, entonces

$$\begin{aligned}
Coste(E(R) \vartriangle E(P)) &= \sum_{(v_i,v_j)\in E(R)\vartriangle E(P)} c_{i,j} \\
&= \sum_{(v_i,v_j)\in E(R)\setminus E(P)} c_{i,j} + \sum_{(v_i,v_j)\in E(P)\setminus E(R)} c_{i,j} \\
&= \sum_{(v_i,v_j)\in E(R)} c_{i,j} - \sum_{(v_i,v_j)\in E(R)\cap E(P)} c_{i,j} + \sum_{(v_i,v_j)\in E(R)\setminus E(P)} c_{i,j} \\
&= \sum_{(v_i,v_j)\in E(R)} c_{i,j} - \sum_{i=0}^{m-1} c_{2i+1,2i+2} + \sum_{i=0}^{m-1} c_{2i+2,2i+3} \\
&= Coste(E(R)) - g(P).
\end{aligned}$$

En cuanto a la parte (b), sea k el mayor de los índices tales que $g((v_1, v_2, \ldots, v_{2k+1}))$ es mínimo. Se denota por $Q = (v_{2k+1}, v_{2k+2}, \ldots, v_{2m-1}, v_1, v_2, \ldots, v_{2k+1})$. Esta nueva secuencia de vértices es un camino alterno cerrado con $E(P) = E(Q)$. Para probar que Q es adecuado basta ver que dado $i = k+1, \ldots, m$

$$g((v_{2k+1}, v_{2k+2}, \ldots, v_{2i+1})) = g((v_1, v_2, \ldots, v_{2i+1})) - g((v_1, v_2, \ldots, v_{2k+1})) > 0,$$

por la propiedad pedida para k. Por tanto, para todo $i = 1, \ldots, k$ se tiene que

$$\begin{aligned} g((v_{2k+1}, \ldots, v_{2m}, v_1, \ldots, v_{2i+1})) &= g((v_{2k+1}, v_{2k+2}, \ldots, v_{2m+1})) + g((v_1, v_2, \ldots, v_{2i+1})) \\ &\geq g((v_{2k+1}, v_{2k+2}, \ldots, v_{2m+1})) + g((v_1, v_2, \ldots, v_{2k+1})) \\ &= g(P) > 0, \end{aligned}$$

por la definición de k. Por tanto, se concluye que Q es adecuado. □

En estas condiciones, se define el *algoritmo de Lin-Kernighan*, que en cada iteración busca un camino alterno sobre el grafo hasta encontrar alguno que verifique las condiciones del Lema 2.8. Una versión en pseudocódigo del *algoritmo de Lin-Kernighan*, presente en Korte y Vygen [2006], se ilustra en el Algoritmo 3.

Los parámetros p_1 y p_2 del algoritmo establecen la longitud máxima de los caminos alternos que el algoritmo prueba. A mayores valores de los parámetros, más caminos alternos probará sobre la ruta. Shen Lin y Brian Kernighan, autores del algoritmo propusieron los valores $p_1 = 5$ y $p_2 = 2$, pues son los más pequeños que garantizan que el algoritmo alcanza resultados tan buenos o mejores que el algoritmo *3-Opt*, tal y como establece el siguiente resultado.

Teorema 2.9 (Lin & Kernighan (1973))**.** *El Algoritmo 3:*

(a) para $p_1 = \infty$ y $p_2 = \infty$ encuentra, si existe, un camino alterno P cerrado y adecuado tal que $E(R) \vartriangle E(P)$ es una ruta.

(b) para $p_1 = 5$ y $p_2 = 2$ devuelve una ruta imposible de encontrar con el algoritmo k-Opt con $k > 3$.

Demostración. Sea R la ruta que devuelve el algoritmo tras ejecutarse. Entonces el valor de g^* será cero desde que fue efectuado el último cambio sobre la ruta. En el caso $p_1 = p_2 = \infty$, el algoritmo habrá probado todos los posibles caminos alternos adecuados sobre la ruta, lo que concluye que (a) es cierto.

Para el apartado (b), sea R la ruta final y sean $p_1 = 5$ y $p_2 = 2$. Con esos valores el algoritmo habrá probado todos los caminos alternos de longitudes 4 y 6 sobre la ruta. Se supone que existe alguna mejora *2-change* o *3-change* que transforma R en otra ruta R' más corta. Entonces $E(R) \vartriangle E(R')$ forman un camino alterno P cerrado con a lo sumo seis aristas y $g(P) > 0$. Entonces, es posible concluir por el Lema 2.8 que P es adecuado y por tanto el algoritmo lo habría encontrado, por lo que hemos llegado a una contradicción. □

Algoritmo 3 Algoritmo de Lin-Kernighan

Entrada: Una instancia $\mathcal{I} = (G, C)$, con $G = (V, E)$ del TSP. Dos parámetros $p_1, p_2 \in \mathbb{N}$
Salida: Una ruta R

1: Sea R una ruta.
2: $X_0 := V$, $i := 0$ y $g^* := 0$.
3: **si** $V_i = \varnothing$ y $g^* > 0$ **entonces**
4: $E(R) := E(R) \triangle E(P^*)$ y volver a 2.
5: **fin si**
6: **si** $V_i = \varnothing$ y $g^* = 0$ **entonces**
7: $i := \text{mín}\{i - 1, p_1\}$
8: **si** $i < 0$ **entonces**
9: Fin de ejecución.
10: **si no**
11: Volver a 3.
12: **fin si**
13: **fin si**
14: Sea $x_i \in V_i$, $V_i := V_i \setminus \{v_i\}$.
15: **si** i impar, $i \geq 3$, $E(R) \triangle E((v_0, v_1, \ldots, v_{i-1}, v_i, v_0))$ es ruta y $g((v_0, v_1, \ldots, v_{i-1}, v_i, v_0)) > g^*$ **entonces**
16: $P^* := (v_0, v_1, \ldots, v_{i-1}, v_i, v_0)$, $g^* := g(P^*)$.
17: **fin si**
18: **si** i impar **entonces**
19: $V_{i+1} := \{v \in V \setminus \{v_0, v_i\} : (v_i, v) \notin E(R) \cup E((v_0, v_1, \ldots, v_{i-1})), g((v_0, v_1, \ldots, v_{i-1}, v)) > g^*\}$.
20: **fin si**
21: **si** i par, $i \leq p_2$ **entonces**
22: $V_{i+1} := \{v \in V : (v_i, v) \in E(R) \setminus E((v_0, v_1, \ldots, v_i))\}$.
23: **fin si**
24: **si** i par, $i > p_2$ **entonces**
25: $V_{i+1} := \{v \in V : (v_i, v) \in E(R) \setminus E((v_0, v_1, \ldots, v_i)),$
$(v, v_0) \notin E(R) \cup E((v_0, v_1, \ldots, v_i))$, $E(R) \triangle E((v_0, v_1, \ldots, v_i, v, v_0))$ es una ruta$\}$.
26: **fin si**
27: $i = i + 1$, volver a 3.
28: **devolver** R

El texto Korte y Vygen [2006] concluye destacando la efectividad del *algoritmo Lin-Kernighan*, superando con creces la del *3-Opt*. Mientras que el *algoritmo Lin-Kernighan* es al menos tan bueno como el *3-Opt*, el tiempo de ejecución esperado del mismo tomando los parámetros $p_1 = 5$ y $p_2 = 2$ también es favorable, pues reporta una complejidad en tiempo de ejecución de $\mathcal{O}(n^{2.2})$ [Korte y Vygen, 2006]. El problema radica en que la complejidad temporal deja de ser polinomial en el peor caso posible.

Casi todos los algoritmos de búsqueda local se basan en este algoritmo debido a que las soluciones que obtiene suelen ser próximas a la óptima a la vez que el tiempo que requiere su ejecución es razonable en la mayoría de los casos. Sin embargo, otros algoritmos funcionan mejor que Lin-Kernighan en el peor caso, como el *algoritmo de Christofides*.

2.2.4. Algoritmo Or-Opt

El algoritmo *Or-Opt* es uno de los algoritmos más conocidos de la clase CE (*chain exchanges*). A diferencia de los métodos *k-Opt* y *Lin-Kernighan*, que basan sus técnicas de modificación en la eliminación de aristas de la ruta que son reemplazadas por otras dando como resultado una nueva ruta, el algoritmo *Or-Opt* selecciona una cadena de vértices consecutivos y una arista de la ruta, y construye una nueva ruta posicionando la cadena de vértices entre los dos nodos que unía la arista eliminada [Babin et al., 2007].

A lo largo de la ejecución del algoritmo, se prueba con diferentes longitudes de cadena prefijadas al comienzo. Una implementación en pseudocódigo del algoritmo *Or-Opt* se presenta en la Figura 4.

Algoritmo 4 Algoritmo Or-Opt

Entrada: Una instancia $\mathcal{I} = (G, C)$, con $G = (V, E)$ del TSP. Parámetros: longitudes de cadena $l_1, l_2, \ldots, l_s \in \mathbb{N}$
Salida: Una ruta R

```
1: Sea R una ruta
2: para l_t hacer
3:   para v_k, vértice en la ruta R hacer
4:     para e ∈ E(R) hacer
5:       Asignar a R' la ruta resultante de eliminar e y reconectar en su lugar la cadena
         de vértices de R que conforman v_k y los (l_t − 1) siguientes.
6:       si Σ_{(v_i,v_j)∈E(R')} c_{i,j} < Σ_{(v_i,v_j)∈E(R)} c_{i,j} entonces
7:         R := R'
8:         Salir del bucle sobre v_k
9:       fin si
10:    fin para
11:  fin para
12: fin para
13: devolver R
```

En Babin et al. [2007] se presenta a su vez una variante del algoritmo que intercam-

bia los bucles. El algoritmo fija primero un vértice y sobre él va iterando con distintas longitudes de cadena hasta encontrar una y una arista que al introducir la cadena en la posición donde se encontraba la arista eliminada, se obtenga una nueva ruta más corta que la anterior. El pseudocódigo de esta variante se presenta en el Algoritmo 5.

Algoritmo 5 Variante algoritmo Or-Opt: *vertex first*

Entrada: Una instancia $\mathcal{I} = (G, C)$, con $G = (V, E)$ del TSP. Parámetros: longitudes de cadena $l_1, l_2, \ldots, l_s \in \mathbb{N}$
Salida: Una ruta R

Sea R una ruta
para v_k, vértice en la ruta R **hacer**
  para l_t **hacer**
    para $e \in E(R)$ **hacer**
      Asignar a R' la ruta resultante de eliminar e y reconectar en su lugar la cadena de vértices de R que conforman v_k y los $(l_t - 1)$ siguientes.
      si $\sum_{(v_i,v_j)\in E(R')} c_{i,j} < \sum_{(v_i,v_j)\in E(R)} c_{i,j}$ **entonces**
        $R := R'$
        Salir del bucle sobre l_t
      fin si
    fin para
  fin para
fin para
devolver R

Del análisis de ambos códigos se obtiene un coste computacional asociado de orden $\mathcal{O}(n^2)$, siendo n el número de nodos de la instancia.

Capítulo 3

Aplicación de las heurísticas a la resolución de ejemplos generales

En los capítulos anteriores, se han recogido las bases teóricas que permiten contextualizar el problema del viajante (TSP). En el Capítulo 1, se persiguió probar la ya mencionada dificultad de resolución del problema del viajante. En el Capítulo 2, se recopilan los principales algoritmos aplicables a la resolución de instancias del problema del viajante, profundizando en un tipo especial de éstos, las *heurísticas de mejora local*, cuyo estudio es uno de los objetivos de este Trabajo Fin de Grado (ver Sección 2.2).

Con todo este compendio teórico, estamos en condiciones de poner en práctica los distintos algoritmos de resolución vistos, para así poder obtener unas primeras conclusiones y comparativas sobre su comportamiento.

3.1. Procedimiento de las heurísticas de mejora local

El problema del viajante ha sido ampliamente estudiado en la literatura, debido fundamentalmente a la dificultad que entraña su resolución, así como a la multitud de aplicaciones prácticas que presenta. Esto explica que en la actualidad se disponga de una gran variedad de métodos aplicables a la resolución de instancias del problema. Cada uno de estos algoritmos presenta características únicas que determinan o condicionan su forma de proceder. El estudio de este comportamiento permite concluir el nivel de adecuación de cada uno de ellos a resolver instancias con propiedades específicas del TSP.

Esta sección centra su análisis en el estudio de los métodos heurísticos de mejora local. Concretamente, se analizan los algoritmos *2-Opt*, *3-Opt*, *Lin-Kernighan* y *Or-Opr*, ya introducidos en el Capítulo 2. Estos métodos muestran, por lo general, un comportamiento que los hace adecuados para la resolución de gran parte de las instancias del problema.

Las heurísticas de mejora local son consideradas por Korte y Vygen [2006] como la técnica más adecuada para obtener buenas soluciones a buena parte de las instancias del problema del viajante. La mecánica de todos estos algoritmos, tal y como mencionábamos

en el Capítulo 2, consiste en iniciar su ejecución con una ruta de partida (un ciclo hamiltoniano sobre el grafo de la instancia), obtenida a través de cualquier otra técnica (lo aconsejable es que sea una heurística constructiva, por su sencillez), e intentar "mejorarla" (encontrar una ruta más corta mediante la realización de pequeñas modificaciones sobre ella). Para realizar estos cambios sobre la ruta actual, cada algoritmo hace uso de un conjunto acotado de procedimientos y técnicas que caracteriza a cada heurística.

3.1.1. Funcionamiento de los algoritmos k-Opt

Los métodos de la familia *k-Opt*, conforman los algoritmos de búsqueda local de implementación más sencilla. Se ha visto en el Capítulo 2 que la técnica de mejora de la que hacen uso recibe el nombre de *k-change* (ver Definición 2.6) y consiste en suprimir k aristas de la ruta actual reemplazándolas por otras k que dan como resultado una nueva ruta. El valor de k es fijo para cada iteración del algoritmo y dependiente de la heurística de la familia *k-Opt* utilizada. Para el *2-Opt*, $k = 2$; para el *3-Opt* $k = 3$; y así sucesivamente.

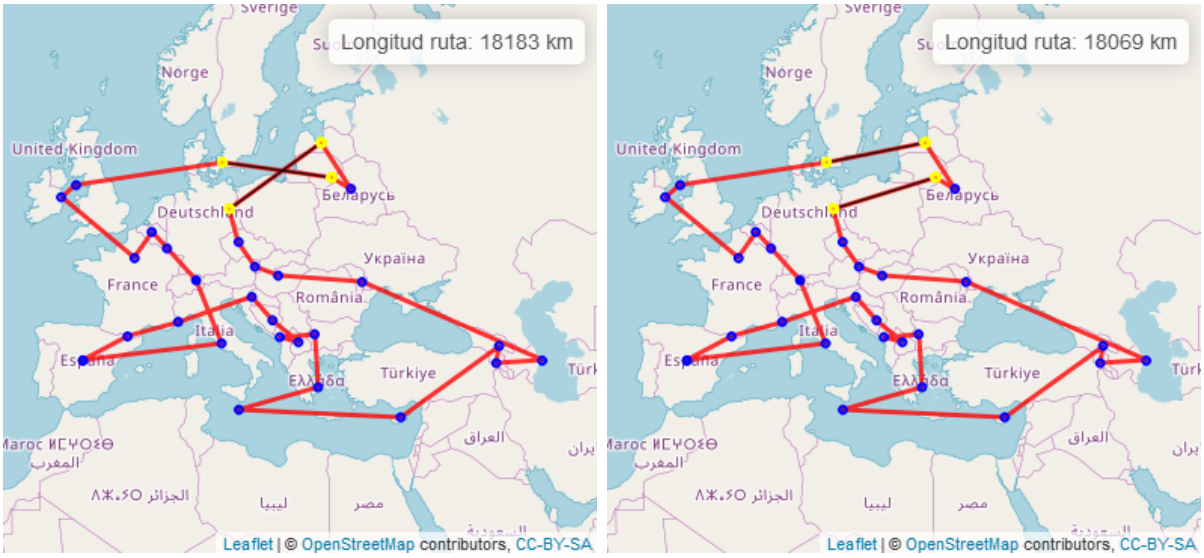
Los algoritmos *k-Opt* son ampliamente utilizados por obtener en general buenas soluciones sin requerir para ello un elevado coste computacional. Sin embargo, en el Capítulo 2 se ha probado la existencia de instancias del problema para las que la ejecución del *2-Opt* puede llegar a requerir un coste computacional de orden exponencial (ver Teorema 2.5), y de la misma forma, mencionábamos que Chandra et al. [1999] generaliza la tesis anterior para cualquier algoritmo de la familia *k-Opt*.

A pesar de hacer uso de una mecánica similar, el comportamiento de cada uno de estos algoritmos difiere del de los demás y por tanto se predispone que alcanzarán soluciones de distintas características y con diferente coste computacional. Esto dependerá del valor de k elegido y, por supuesto, del número de vértices de la instancia y los valores de la matriz de costes asociada.

Las modificaciones más pequeñas son las realizadas por el algoritmo *2-Opt*, pues solo dos aristas de la ruta actual son eliminadas. Este tipo de cambios es más deseable cuando nos aproximamos a la solución óptima, de manera que la ruta requiere de pequeños cambios para mejorar, mientras que grandes modificaciones no harían posible mejorarla. La Figura 3.1 ilustra un *2-change* sobre una instancia del TSP con 30 ciudades.

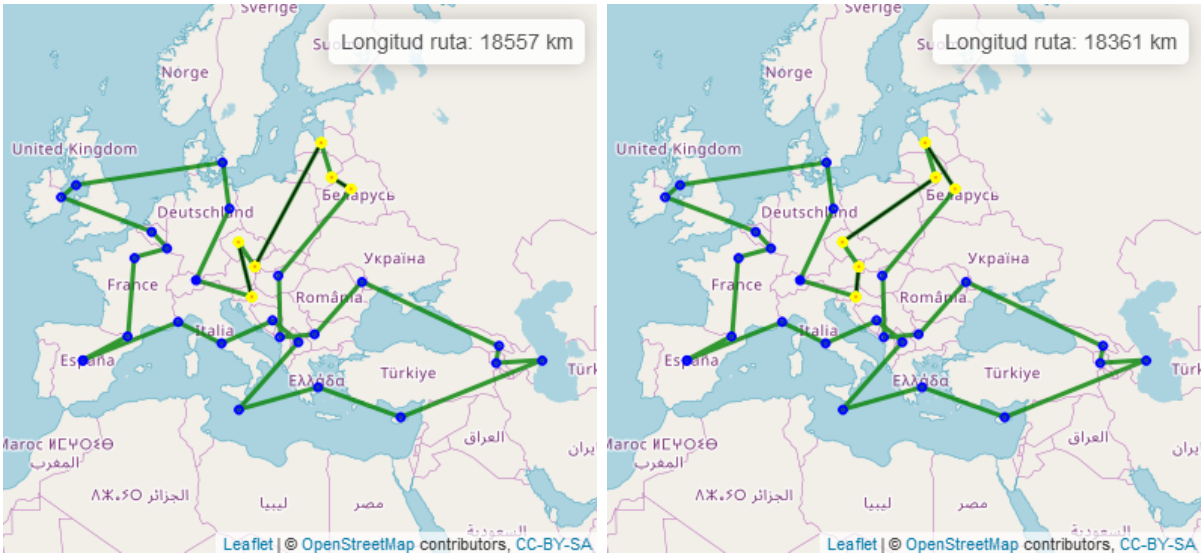
En el caso del *3-Opt*, un *3-change* realizaría modificaciones mayores, pero a su vez menos refinadas que las que puede obtener un *2-change*, lo que predispone a este último como menos preferible en caso de buscar soluciones más próximas a la óptima. La Figura 3.2 ilustra un *3-change* sobre la misma instancia que trata la Figura 3.1.

Esta tendencia continúa para cualquier valor de k. La práctica ha demostrado que a pesar de disponer de rutas iniciales muy alejadas del óptimo, es preferible el uso del *2-Opt*, que aun haciendo modificaciones más leves, suele mejorar la ruta de forma más rápida y alcanza al final de su ejecución rutas más cortas por el pequeño alcance de sus modificaciones.



(a) Ruta actual (b) Ruta mejorada

Figura 3.1: Ilustración de un *2-change* durante la ejecución del *2-Opt* sobre una instancia con 30 ciudades. En negro y amarillo, aristas y nodos afectados respectivamente

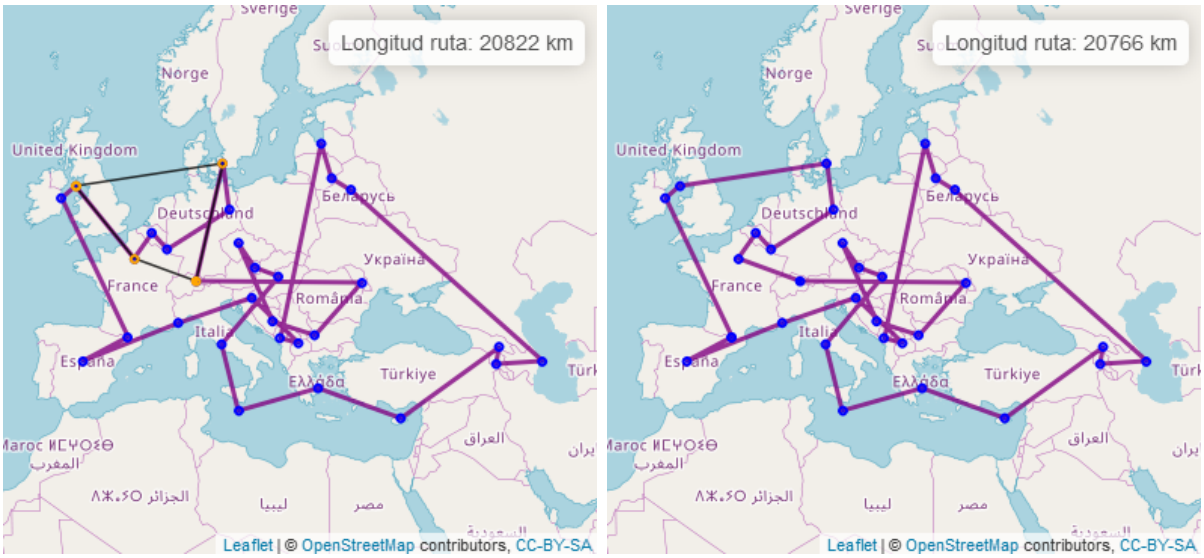


(a) Ruta actual (b) Ruta mejorada

Figura 3.2: Ilustración de un *3-change* durante la ejecución del *3-Opt* sobre una instancia con 30 ciudades. En negro y amarillo, aristas y nodos afectados respectivamente

3.1.2. Funcionamiento del algoritmo de Lin-Kernighan

A diferencia de los algoritmos de la familia *k-Opt*, el algoritmo de *Lin-Kernighan* basa sus modificaciones en la variabilidad en la magnitud de las mismas. Así, basa sus modificaciones en el concepto de camino alterno (ver Definición 2.7), que al combinarse con la ruta actual, genera una modificación cuyo alcance es proporcional a la longitud del camino. Esta variación en la magnitud de los cambios hace a este algoritmo adaptarse mejor a las circunstancias de la instancia, llegando a alcanzar soluciones tan buenas como haría el algoritmo *2-Opt* o el *3-Opt*, como se ha mencionado en el Capítulo 2.



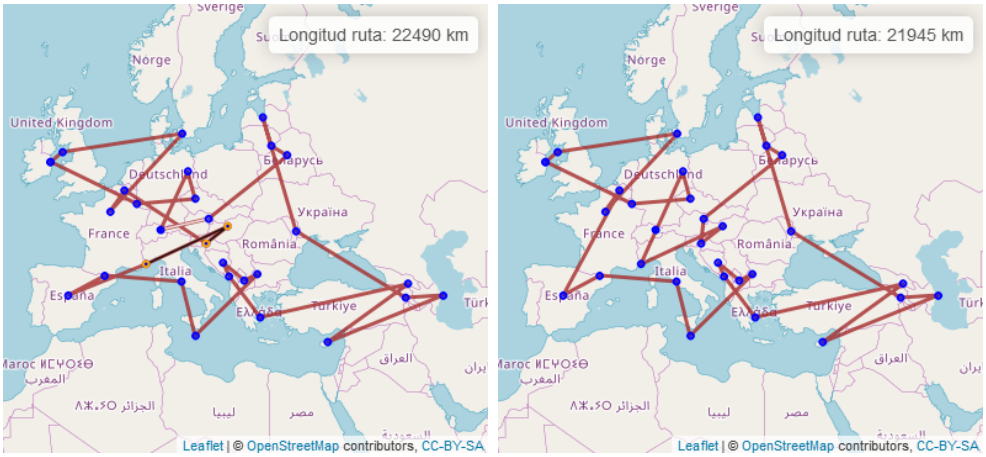
(a) Ruta actual (b) Ruta mejorada

Figura 3.3: Ilustración de una mejora durante la ejecución del algoritmo *Lin-Kernighan* sobre una instancia con 30 ciudades. En naranja y negro, el camino alterno encontrado

La Figura 3.3 ilustra una modificación efectuada por el algoritmo de *Lin-Kernighan* sobre la ruta de una instancia con 30 ciudades. En negro está pintado el camino alterno encontrado en (a). En (b) se muestra el resultado de combinar el camino alterno encontrado con la ruta actual, obteniendo una nueva ruta.

3.1.3. Funcionamiento del algoritmo Or-Opt

Por último, se analiza el procedimiento de mejora usadas por el algoritmo *Or-Opt* (ilustrado en la Figura 3.4). Este método dota al usuario de un mayor control en el procedimiento ya que es éste quien decide las longitudes de cadena que el algoritmo irá probando a lo largo del proceso, así como el orden en que se probarán. En caso de ser conocedor de la capacidad de mejora que tiene la ruta de partida, podrá prever las longitudes que harán más eficiente la ejecución.



(a) Ruta actual (b) Ruta mejorada

Figura 3.4: Ilustración de una mejora con longitud de cadena 3 durante la ejecución del algoritmo *Or-Opt* (ver Algoritmo 4) sobre una instancia con 30 ciudades. En negro y naranja la cadena de vértices, y en blanco la arista que se sustituye por la cadena

Entre las ventajas de este método está su gran flexibilidad, capaz de resolver rápidamente instancias cuya ruta de partida difiere mucho de la solución óptima, así como su reducida complejidad computacional siendo recomendable su aplicación sobre instancias grandes. Como principal desventaja está la necesaria elección correcta de las longitudes de cadena, pues en caso contrario el algoritmo puede resultar ineficiente.

La Figura 3.6 ilustra las soluciones que obtienen los cuatro algoritmos expuestos en esta sección al ser ejecutados sobre el mismo supuesto con 30 ciudades y comenzando en la misma ruta de partida (ilustrada en la Figura 3.5). Obsérvese que la ruta del algoritmo *3-Opt* se entrecruza en la zona de Bélgica. El algoritmo no es capaz de mejorarla ya que para romper el cruce es necesario eliminar las dos aristas que se cortan entre sí, manteniendo las demás, lo cual no es posible (sí lo sería con un *2-change*).

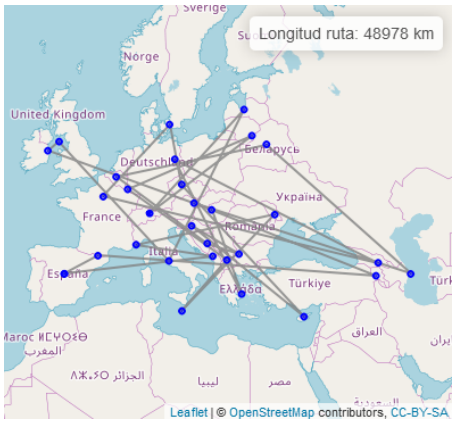
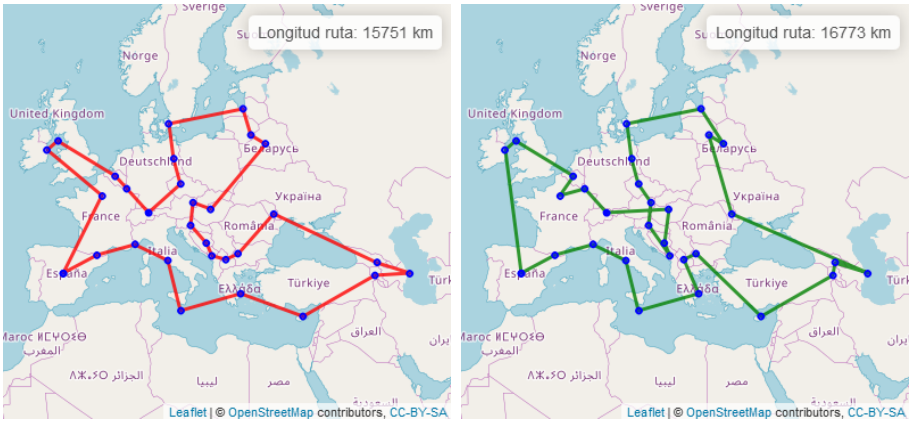
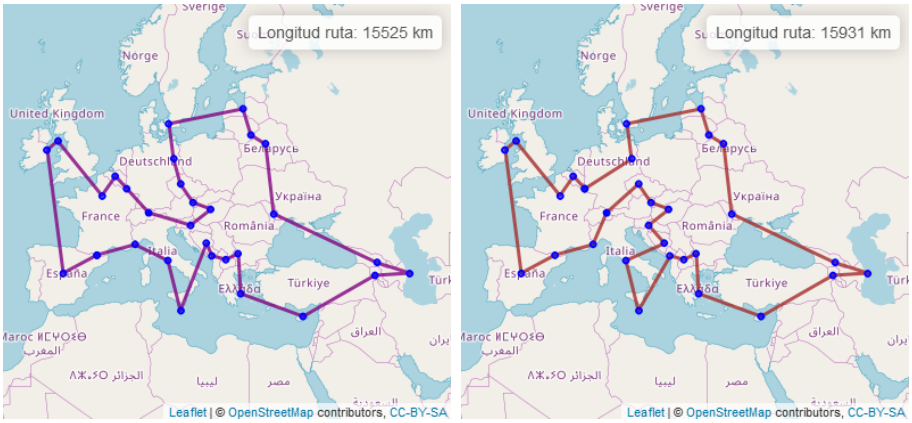


Figura 3.5: Ruta de partida para la ejecución de los algoritmos de búsqueda local expuestos en esta sección, cuyos resultados finales se ilustran en la Figura 3.6



(a) *2-Opt* (b) *3-Opt*



(c) *Lin-Kernighan*, $p_1 = 5$, $p_2 = 2$ (d) *Or-Opt*, $longitudes = (12, 6, 3, 2, 2, 1, 1, 1)$

Figura 3.6: Rutas finales obtenidas de la ejecución de los cuatro algoritmos ilustrados sobre una misma instancia con 30 ciudades y con la ruta inicial que muestra la Figura 3.5

3.2. Aplicación de los algoritmos a la resolución de instancias generales del TSP

A lo largo de esta sección se persigue analizar el comportamiento de los algoritmos más importantes para la resolución del problema del viajante (TSP) en base a ejemplos generales. Se aplica para ello cada uno de los algoritmos a la resolución de una serie de instancias con diferente número de ciudades.

Los algoritmos que se analizan a lo largo de esta sección son el *2-Opt*, *vecinos más próximos*, *inserción más lejana*, *vecinos más próximos repetitivo*, *inserción más barata*, *inserción más cercana*, *Lin-Kernighan*, *Concorde* y *simulated annealing*. Haremos uso de implementaciones en `R` de cada uno de ellos, las cuales encontramos en la librería `TSP`, a excepción del *simulated annealing*, que se encuentra en la librería `optim`.

Cabe destacar que este estudio fue implementado sobre un ordenador *ACER Aspire 5 A515-51G-751G* con sistema operativo *Windows 10 Home*, procesador *Intel Core i7-7500U con velocidad de entre 2.7 GHz a 3.5 GHz* y memoria RAM de 8GB. Se realiza conjuntamente por María Zorita Mínguez y por mi en el marco de las investigaciones que la Universidad de Valladolid lleva a cabo con *Boeing Research & Technology Europe*.

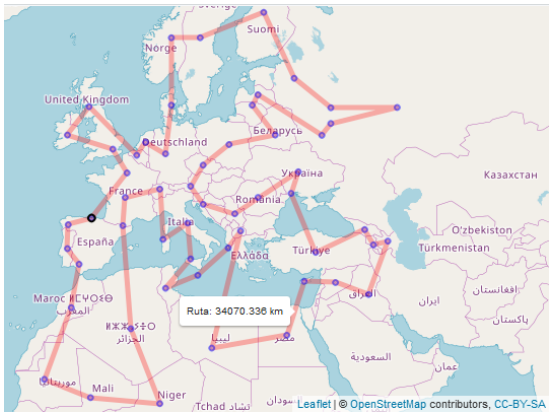
Se toman tres instancias sobre conjuntos fijos de ciudades sobre la Tierra. La primera se compone de 55 ciudades distribuidas por Europa y el Norte de África. En la segunda se añaden más ubicaciones alcanzando 502 nodos, y en la tercera se dispone de 1200.

El objetivo es estudiar el comportamiento de los principales algoritmos de aplicación al TSP en diferentes supuestos que incluyen instancias de distintos tamaños.

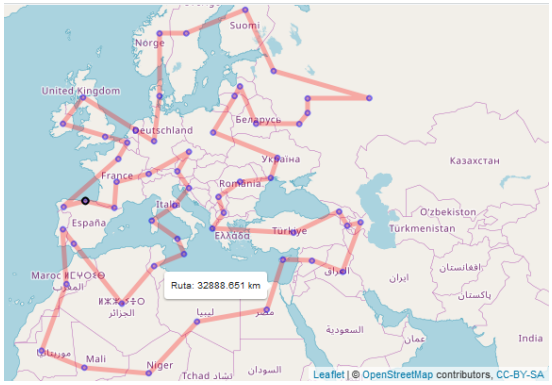
Método	**55 ciudades**		**502 ciudades**		**1200 ciudades**	
	Longitud (km)	**Tiempo (seg)**	**Longitud (km)**	**Tiempo (seg)**	**Longitud (km)**	**Tiempo (seg)**
Inserción más cercana	34.070	**0,00**	129.688	0,79	181.704	9,59
Inserción más lejana	32.888	**0,00**	105.201	0,89	160.878	7,47
Inserción más barata	35.410	0,03	115.215	0,58	170.368	6,42
Inserción aleatoria	32.084	**0,00**	107.423	0,02	162.013	**0,02**
Vecinos más próximos	34.796	**0,00**	130.530	**0,00**	187.372	0,05
Vecinos más próximos repetitivo	33.783	0,07	118.415	8,23	185.668	84,14
2-Opt	36.857	**0,00**	108.908	0.44	160.585	8.37
Lin-Kernighan	**30.417**	0,19	100.692	3,37	148.663	17,92
Concorde	**30.417**	0,50	**98.438**	13,74	**147.638**	61,59
Simulated annealing 40.000 iteraciones	32.706	2,70	105.113	4,91	160,878	13,54
Simulated annealing 100.000 iteraciones	32.242	9,32	104.987	10,60	160,846	27,64
Simulated annealing 1.000.000 iteraciones	32.230	70,83	104.512	105,89	160,479	164,34

Tabla 3.1: Longitud de la ruta obtenida y tiempo de ejecución requerido por cada algoritmo estudiado en base a los tres ejemplos considerados

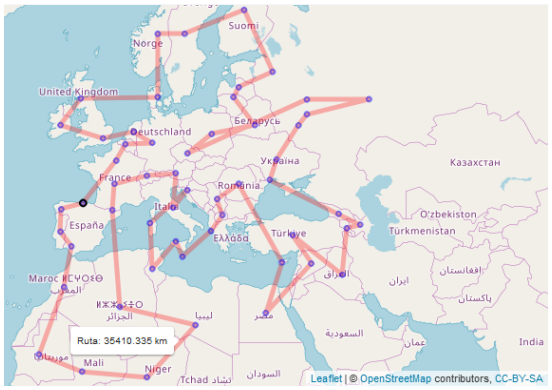
La Tabla 3.1 presenta los resultados de las ejecuciones llevadas a cabo en el estudio. Por un lado, los resultados arrojados por los algoritmos constructivos avanzan la eficacia del método de *inserción más lejana*, que alcanza las mejores soluciones en todos los supuestos, entre este conjunto de algoritmos. Sin embargo, es superado ampliamente en términos de tiempo de ejecución, por el método de *vecinos más próximos* (150 veces más lento sobre la instancia de 1200 ciudades) y el de *inserción arbitraria* (373 veces más lento sobre la instancia de 1200 ciudades).



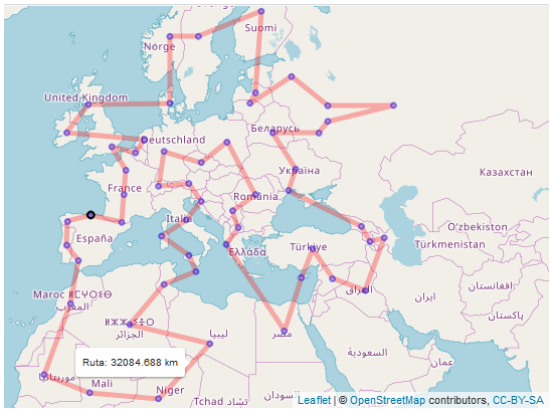
Inserción más cercana



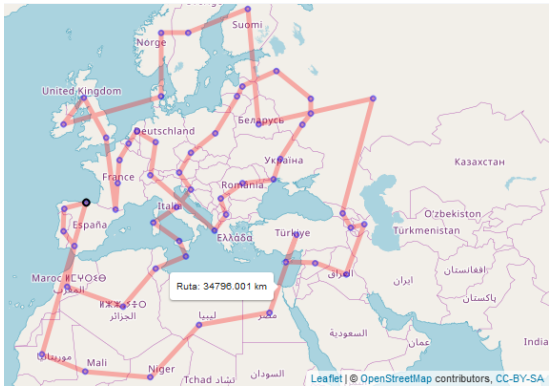
Inserción más lejana



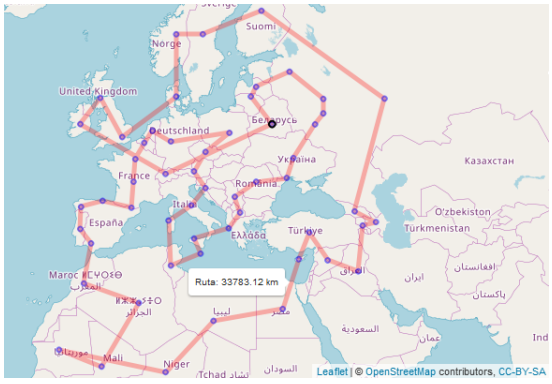
Inserción mas barata



Inserción arbitraria

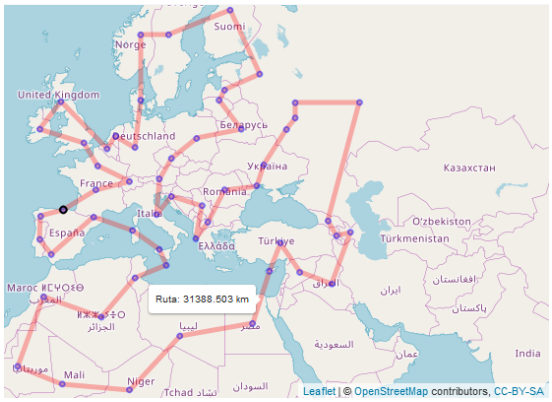


Vecinos más próximos

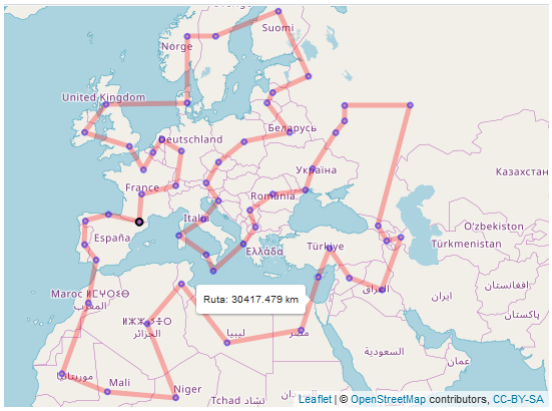


Vecinos más próximos repetitivo

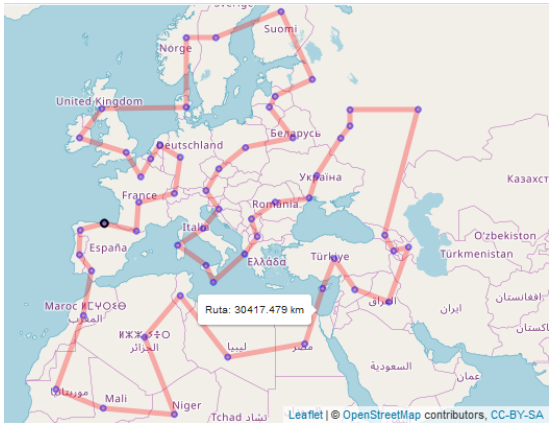
Figura 3.7: Rutas encontradas sobre la instancia de 55 ciudades. En negro, la ciudad sobre la que comienza y termina la ruta encontrada (i).



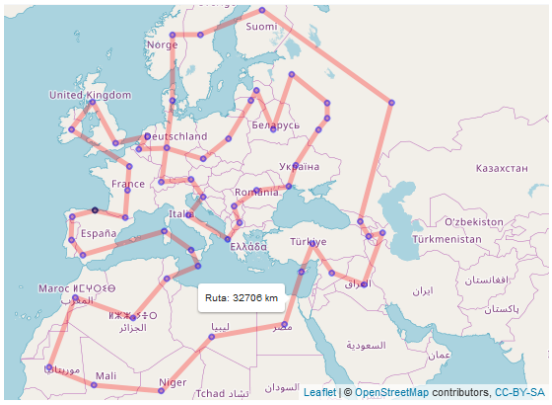
2-Opt



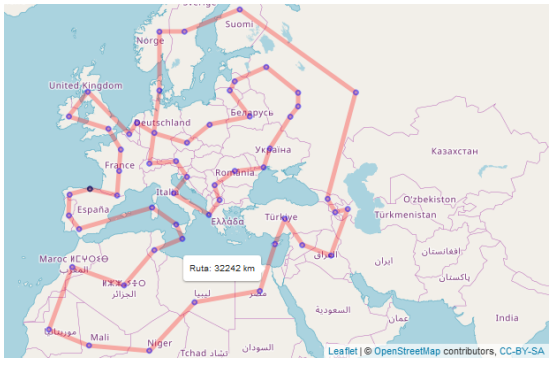
Lin-Kernighan



Concorde



Simulated annealing, 40.000 iteraciones



Simulated annealing, 100.000 iteraciones

Simulated annealing, 1.000.000 iteraciones

Figura 3.8: Rutas encontradas sobre la instancia de 55 ciudades. En negro, la ciudad sobre la que comienza y termina la ruta encontrada (ii).

El algoritmo de *inserción arbitraria* combina buenos tiempos de ejecución con resultados próximos a la mejor solución alcanzada en todos los casos. Por contra, la variabilidad de los resultados arrojados por este algoritmo es muy grande por estar sujeto su comportamiento a una alta aleatoriedad.

En cuanto al resto de métodos, se toma una ordenación arbitraria de los datos considerados como ruta de partida (la misma para todos), con el fin de estudiar el comportamiento de éstos partiendo de una ruta que debería estar muy alejada, en términos de longitud, de la que podría obtenerse haciendo uso de cualquier método constructivo. Los resultados indican una fuerte debilidad del *simulated annealing* con respecto al resto, que mejora los resultados del *2-Opt*, pero a diferencia de éste, requiere un tiempo de ejecución elevado. Este hecho, hace sospechar que requiere de la ejecución de un método constructivo previo para obtener una ruta inicial aceptable. En cuanto al resto, destacan, en términos de longitud de la ruta obtenida, los algoritmos *Lin-Kernighan* y *Concorde*, siendo éste último algo más eficaz ($2,23\,\%$ mejor sobre la instancia de 502 y $0,68\,\%$ sobre la de 1200 ciudades) pero requiriendo para ello mucho más tiempo (4 veces más en la instancia de 502 ciudades y algo más de 3 sobre la de 1200, pero superando el minuto).

Se observa por tanto la supremacía de los algoritmos *Lin-Kernighan* y *Concorde* sobre instancias pequeñas, cuyos tiempos de ejecución son razonables. Sin embargo, frente a instancias algo más grandes parece recomendable dejar a un lado *Concorde* para hacer uso de *Lin-Kernighan* debido a las notables diferencias observadas en tiempos de ejecución.

En resumen, se puede concluir en base a este estudio que, tal y como se menciona en Korte y Vygen [2006], las heurísticas de mejora local muestran un buen comportamiento en la práctica, encontrando soluciones razonables sin requerir, por lo general, un coste computacional de orden muy elevado. Es por ello que han sido ampliamente estudiadas y tratadas por multitud de autores, de manera que contemplar las diversas contribuciones y perspectivas de cada uno de ellos resulta inabarcable en el marco de este trabajo fin de grado.

Capítulo 4

Aplicación al problema de la reordenación de los datos de seguimiento

Hasta ahora, este trabajo se ha centrado en los fundamentos teóricos del problema del viajante a modo de estado del arte del problema que, como se menciona en la introducción, es objeto de estudio de esta memoria: el *problema de la reordenación de los datos de seguimiento ADS-B de una trayectoria*.

Como ya se ha comentado, el TSP sirve de modelo sobre el cual se construye una solución para el supuesto anterior. Este capítulo tiene como objetivo principal la exposición de esta propuesta.

4.1. El proyecto AIRPORTS

En línea con lo expuesto en la introducción, Europa ha visto como en los últimos años ha ido aumentando progresivamente la densidad del tráfico aéreo continental, haciendo necesaria una profunda renovación de sus sistemas de gestión en la que se apueste por nuevas tecnologías adaptadas a las actuales necesidades del sector. Así, surgen iniciativas como *Single European Sky (SES)*, que entre otras medidas, trajo consigo la transición hacia la ya citada tecnología *Automatic Dependent Surveillance-Broadcast* (ADS-B).

Sin embargo, el uso de esta tecnología implica gestionar las ingentes cantidades de datos de seguimiento que se producen (cada aeronave en vuelo emite hasta dos mensajes ADS-B por segundo). Sobre esta problemática surge la actual colaboración de la Universidad de Valladolid con *Boeing Research and Technology Europe*, que pretende construir una plataforma *bigdata* para llevar a cabo la gestión del tráfico aéreo (ATM) y que actúe como centro de datos, englobando los procesos desde la ingesta de datos "en crudo" emitidos por cada vuelo hasta el refinamiento de los mismos que de lugar a información con valor para el usuario final. La descripción de la arquitectura de esta plataforma puede encontrarse en Álvarez Esteban et al. [2017].

Así, nace el proyecto AIRPORTS como un consorcio formado por un conjunto de empresas y *Organismos Públicos de Investigación* (OPI) liderados por *Boeing Research and Technology Europe* con el propósito de desarrollar nuevas tecnologías aplicables a la aeronáutica que mejoren la eficiencia en las operaciones ATM. Se estudia cómo los datos de seguimiento ADS-B procedentes de diferentes proveedores pueden ser combinados con otros mejorando la calidad (veracidad) y siendo más relevantes como entrada para los sistemas ATM. Estos datos, permiten la reconstrucción de trayectorias de vuelos desde el despegue hasta el aterrizaje gracias a la continua emisión de mensajes ADS-B por parte de la aeronave que incluyen su geolocalización.

4.2. El problema de la reordenación de los datos de seguimiento

Como ya se adelantaba en la introducción, la tecnología ADS-B presenta problemas de alineamiento temporal en los datos que produce, generando errores en el orden en que los sistemas ATM disponen los mensajes ADS-B asociados a una misma trayectoria. Esta problemática tiene como principales causas la transmisión de los mensajes a través del aire, que puede desencadenar retardos en las comunicaciones, y a la no sincronización temporal entre las distintas redes de sensores terrestres que nutren de datos a los sistemas ATM.

(a) Vuelo Madrid-Mayorca (b) Vuelo Madrid-Asturias

Figura 4.1: Segmento de dos trayectorias reconstruida con la librería `leaflet` de `R` a partir de las señales ADS-B recibidas

La Figura 4.1 ilustra dos ejemplos de los efectos que produce esta problemática sobre la posterior reconstrucción de las trayectorias. Obsérvese como, la línea roja, une los puntos, que simbolizan los mensajes ADS-B, ordenados por su marca temporal, que al ser añadida por los receptores cuando reciben el mensaje y no por la aeronave cuando los emite, coincide con el orden de recepción de los mismos.

De manera preliminar, podemos plantear el supuesto a modelar de la forma siguiente:

supongamos un avión que realiza una trayectoria desde un aeródromo A hasta un aeródromo B, enviando n mensajes ADS-B en el transcurso de su realización que son etiquetados con *timestamps* $t_1 < t_2 < \ldots < t_n$, siendo t_i la marca temporal del mensaje i para todo $1 \leq i \leq n$. El **problema de reordenación de los mensajes de una trayectoria** consiste en encontrar una permutación circular π sobre $\{1, \ldots, n\}$ tal que para cada par de mensajes $1 \leq i < j \leq n$ se tiene que el mensaje $\pi(i)$ fue enviado antes que el mensaje $\pi(j)$.

Nótese que, por lo general, cualquier desorden producido (i.e. dos mensajes i, j tal que aunque i se envió antes que j sus marcas temporales verifican que $t_j < t_i$) produce una trayectoria más larga que la realmente recorrida por la aeronave. Es por ello que en la mayoría de los casos (salvo excepciones relativas a maniobras de los pilotos en los aeropuertos esperando permisos desde el control para aterrizar), la solución a este problema coincide con la reordenación de los mensajes que se corresponda con la trayectoria más corta. Así, suponiendo que la reordenación correcta es aquella que minimiza la longitud del camino que atraviesa todos los puntos ordenados, se plantea la Definición 4.1, que contiene una aproximación más formal al problema.

Definición 4.1. *Sea $G = (V, E)$ un grafo completo con $V = \{v_1, v_2, \ldots, v_n\}$ como conjunto de vértices y $E = \{(v_i, v_j) \,|\, v_i, v_j \in V, i \neq j\}$ como conjunto de aristas; y sea $C = (c_{ij})$ la matriz $n \times n$ tal que $c_{ij} \geq 0$ es la longitud asociada a la arista (v_i, v_j), y $c_{ii} = 0$ si $1 \leq i \leq n$. Se define el* ***problema de la reordenación de puntos de trayectorias*** *como el de encontrar una permutación $\pi^* \in \Pi$, donde Π denota el conjunto de permutaciones circulares sobre el conjunto $\{1, 2, \ldots, n\}$, tal que*

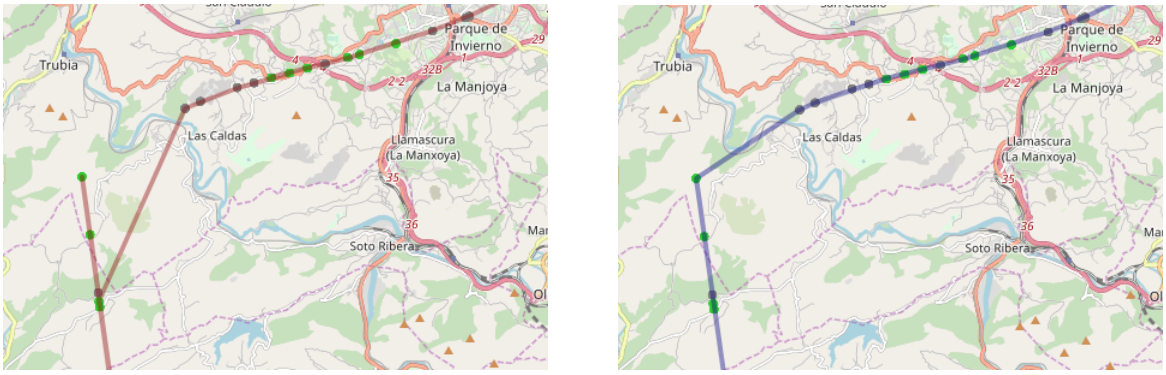
$$\sum_{i=1}^{n} c_{\pi^*(i)\pi^*(i+1)} = \min_{\pi \in \Pi} \sum_{i=1}^{n} c_{\pi(i)\pi(i+1)}.$$

Por tanto, el problema consiste en encontrar un camino hamiltoniano (no cerrado) de mínima longitud. Nótese que este problema así planteado pasa a ser el problema del viajante añadiendo una ciudad artificial que una el nodo final del camino solución con el inicial. Además, para no alterar la solución obtenida, se fija la longitud de todas las aristas que empiezan o acaban en el nodo artificial como 0. De esta forma, se definen las instancias del problema de reordenación de mensajes asociados a trayectorias en términos de las del problema del viajante de la forma siguiente:

Definición 4.2. *Se define una* ***instancia $\mathcal{R}$ del problema de reordenación de los mensajes ADS-B de una trayectoria*** *como un par (G, C) tal que $G = (V, E)$ es un grafo completo con $V = \{v_1, v_2, \ldots, v_n\}$ y $C = (c_{ij})$ la matriz de costes, y tal que existe una instancia $\mathcal{I}' = (G', C')$ del TSP tal que $G' = (V', E')$ con $V' = V \cup \{v_{n+1}\}$ y $C' = (c'_{ij})_{1 \leq i,j \leq n+1}$ dada por*

$$c'_{ij} = \begin{cases} c_{ij} & si \qquad 1 \leq i, j < n+1 \\ \\ 0 & si \quad i = n+1 \ \acute{o} \ j = n+1 \end{cases}$$

De esta forma, $\mathcal{R}$ se identifica con $\mathcal{I}'$ de forma que a la hora de resolver $\mathcal{R}$, basta con obtener una solución para $\mathcal{I}'$, de tal manera que a la ruta obtenida para ésta se le elimina la ciudad artificial v_{n+1} así como las dos aristas de la ruta que conectaban con ella. El camino resultante será por tanto un camino hamiltoniano sobre G cuya longitud, además, es la misma que la de la ruta obtenida sobre G', pues el coste de las dos aristas que han sido eliminadas era, por definición, nulo.



(a) Ruta sin ordenar: 511,99 km

(b) Ruta ordenada: 452,95 km

Figura 4.2: Segmento de una trayectoria Madrid-Asturias sobre la que se ha aplicado una reordenación con el algoritmo *2-Opt*

La Figura 4.2 ilustra la forma de proceder de el modelo presentado en esta sección. La imagen (a) muestra un conjunto de puntos de la trayectoria mal ordenados mientras que en (b) se muestra la nueva disposición, obtenida tras aplicar este modelo resolviendo la instancia del TSP resultante mediante el algoritmo *2-Opt*.

4.3. La técnica de las ventanas de ejecución

Como hemos visto en la sección previa, el modelo de resolución del problema de reordenación de los mensajes ADS-B de una trayectoria parte del TSP. De esta forma, los algoritmos aplicables a la resolución de instancias del problema del viajante también lo serán al anterior.

Utilizar el TSP como modelo conlleva un inconveniente principal, y es que como se ha visto en el Capítulo 1, el problema del viajante es uno de los problemas catalogados como $\mathcal{NP}$*-hard*, lo que implica la no existencia de algoritmos de complejidad computacional polinomial que garanticen encontrar la solución óptima. Esto complica por tanto el planteamiento para trayectorias largas (por ejemplo un vuelo Madrid-Moscú), con un elevado número de mensajes ADS-B, pues el tiempo de ejecución de los algoritmos denominados exactos (que garantizan la solución óptima) se eleva de forma notable, siendo necesario optar por algoritmos de aproximación, entre los que se encuentran las *heurísticas de mejora local*, estudiadas en la Sección 2.2.

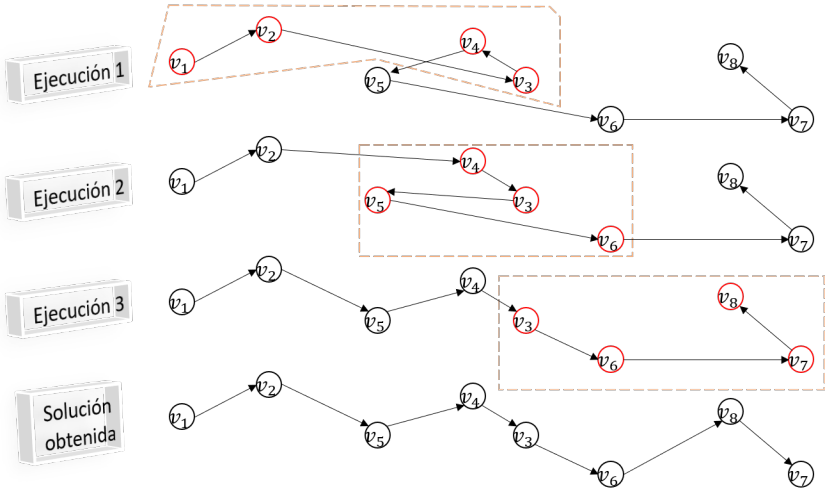


Figura 4.3: Ejemplo de aplicación de la técnica de las ventanas de ejecución con $\tau = 4$ y $\sigma = 2$

Sin embargo, fruto de las investigaciones de la Universidad de Valladolid con *Boeing Research & Technology Europe* surge un método de reducción del coste requerido para la computación de estas reordenaciones sobre trayectorias con una gran cantidad de mensajes ADS-B asociados, que recibe el nombre de *técnica de las ventanas de ejecución*.

Esta estrategia, fija dos enteros $\tau > 2$ y σ tal que $0 \leq \sigma < \tau$. Dada una instancia del problema de reordenación de los puntos de una trayectoria $\mathcal{R} = (G, C)$, donde $G = (V, E)$ es un grafo con $V = \{v_1, v_2, \ldots, v_n\}$, y $C = (c_{ij})$ la matriz $n \times n$ de costes, se definen los subconjuntos de vértices $V_k = \{v_{i_k}, v_{i_k+1}, \ldots, v_{j_k}\}$ siendo $j_k = \text{máx}\{i_k + \tau, n\}$, $i_k = j_{k-1}$, $i_1 = 1$ para todo $k \in \mathbb{N}$. Se construyen entonces los grafos $G_k = (V_k, E_k)$, donde $E_k = \{(v_a, v_b) : v_a, v_b \in V_k\}$, y la matriz $C_k = (c_{ab})_{i_k \leq a, b \leq j_k}$. De esta forma, los pares (G_k, C_k) para cada $k \in \mathbb{N}$ forman nuevas instancias del problema de reordenación de los mensajes asociados a una trayectoria con un máximo de $\tau + 1$ puntos cada una. Al parámetro τ lo denotaremos por *tamaño de la ventana* mientras que a σ lo haremos por *solapamiento*.

Así, obtenemos un conjunto de soluciones $\{R_1, R_2, \ldots, R_s\}$ donde R_k es un camino hamiltoniano en G_k para todo $1 \leq k \leq s$, y a partir de las cuales podemos derivar un camino hamiltoniano R para el grafo G. Obsérvese que, si $\sigma > 0$, cada vértice del problema original se encontrará en más de un subgrafo, e igualmente, cada arista puede aparecer en varios subgrafos. En este caso, formará parte de la solución sobre G si aparece en la solución sobre la instancia (G_m, C_m) tal que

$$m = \text{máx}\{l : \cap_{i=1}^{l} E_i \neq \varnothing\}.$$

De esta forma, el camino hamiltoniano en construcción R sobre G vendrá dado por

$$E(R) = \{e \in E : \text{ si } m = \text{máx}\{k : e \in E_k\}, e \in E(R_m)\}$$

La Figura 4.3 ilustra el funcionamiento de esta técnica. Obsérvese como el solapamiento entre las ventanas hace posible integrar la ordenación obtenida por las distintas ventanas.

Algoritmo	**IBE04HT**		**IBE04NL**		**IBE0519**		**IBE05DK**		**RYR9KY_4CA97C**	
de resolución	Long	Tmp	Long	Tmp	Long	Tmp	Long	Tmp	Long	Tmp
Sin aplicar algoritmo	511,99	-	659,99	-	682,84	-	602,47	-	346,98	-
2-Opt, sin ventanas	452,95	0,42	570,42	0,86	600,38	0,66	558,69	0,36	299,99	0,04
2-Opt, $\tau = 100, \sigma = 20$	452,95	0,04	570,42	0,00	600,38	0,00	558,69	0,00	299,99	0,00

Tabla 4.1: Comparativa de la técnica de ventanas con respecto a su no utilización en longitud de la solución obtenida (Long) en kilómetros y tiempo de ejecución (Tmp) en segundos

Existe una variante de esta técnica que abarca dos tamaños de ventanas, así como dos valores de solapamientos aplicados según la fase en que se encuentre el vuelo. Su uso está pensado para adaptar la técnica a los momentos en que la aeronave se encuentra cerca del aeropuerto de partida o destino, cuando la trayectoria es más irregular debido al mayor número de maniobras que el piloto debe efectuar y a la menor velocidad de la aeronave.

La Tabla 4.1 muestra los resultados de aplicación de la técnica de *ventanas de ejecución*, que expone esta sección, utilizando el algoritmo *2-Opt* sobre varias trayectorias reales. Obsérvese como, sin aumentar la longitud de la solución obtenida, sí se reduce notablemente el tiempo de ejecución (nunca por encima de la décima de segundo usando la técnica).

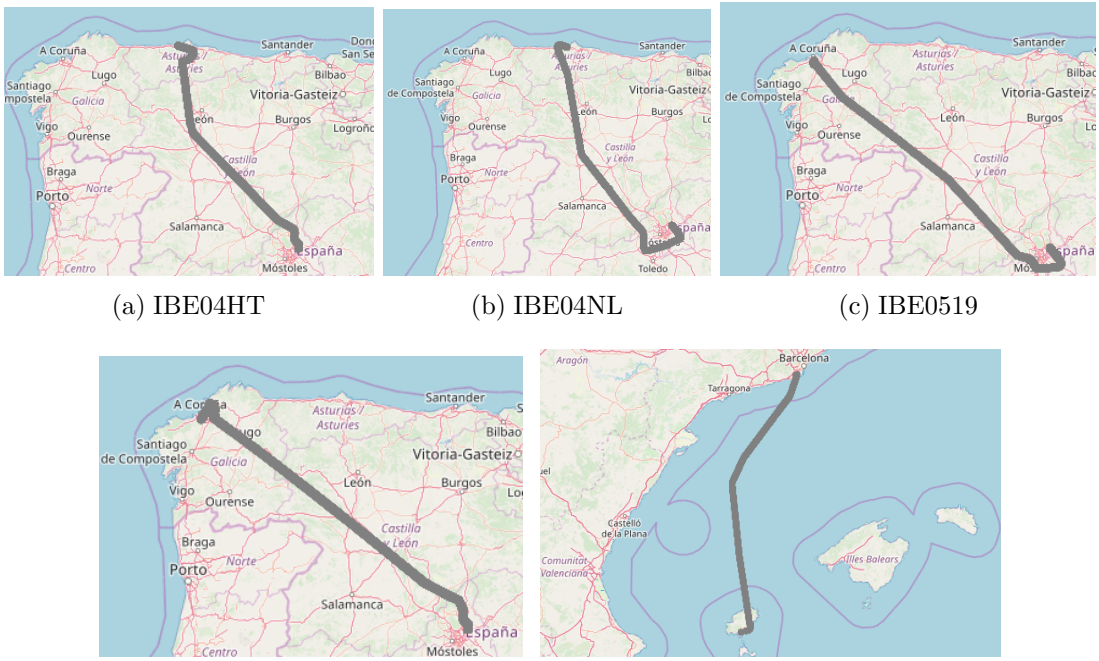
Para las ejecuciones, se han tomado como referencia cinco conjuntos de datos, suministrados por *Boeing*, correspondientes a los siguientes vuelos:

- **IBE04HT**: Madrid-Asturias con 1.090 mensajes ADS-B.
- **IBE04NL**: Asturias-Madrid con 1.151 mensajes ADS-B.
- **IBE0519**: A Coruña-Madrid con 1.105 mensajes ADS-B.
- **IBE05DK**: Madrid-A Coruña con 1.079 mensajes ADS-B.
- **RYR9KY_4CA97C**: Ibiza-Barcelona con 533 mensajes ADS-B.

4.4. Análisis de los algoritmos aplicados a la reordenación de mensajes ADS-B

Para concluir el capítulo, se ha elaborado un estudio comparativo, en colaboración con María Zorita Mínguez, de una serie de algoritmos del TSP aplicados a la reordenación de los mensajes ADS-B de las trayectorias reales que aparecen en la Tabla 4.1.

El comportamiento de los algoritmos frente a la reordenación de estos mensajes se espera que difiera con respecto del estudio contenido en la Sección 3.2, pues en este último, la ruta inicial de partida se tomó aleatoria resultando diferir notablemente con respecto a la solución óptima. En el presente estudio se toma como solución inicial la dada por el orden temporal actual de los mensajes ADS-B, por lo que las diferencias con respecto a la solución óptima se espera que no sean muy grandes.



(a) IBE04HT (b) IBE04NL (c) IBE0519

(d) IBE05DK (e) RYR9KY_4CA97C

Figura 4.4: Representación de los mensajes ADS-B de cada trayectoria sobre un mapa (elaborado con `Leaflet`)

Algoritmo de resolución	**IBE04HT**		**IBE04NL**		**IBE0519**		**IBE05DK**		**RYR9KY_4CA97C**	
	Long	Tmp	Long	Tmp	Long	Tmp	Long	Tmp	Long	Tmp
Sin aplicar algoritmo	511,99	-	659,99	-	682,84	-	602,47	-	346.98	-
Inserción más cercana	453,22	5,46	588,89	7,24	639,72	5,60	573,73	5,59	300.85	0.78
Inserción más lejana	533,07	5,41	570,51	6,14	600,42	5,97	801,13	4,97	360.63	0.80
Inserción más barata	452,95	3,76	570,42	4,03	600,38	3,89	**558,69**	3,04	**299.99**	0.49
Inserción aleatoria	529,76	**0,05**	576,98	**0,05**	873,01	**0,04**	653,8	0,05	330.80	0.02
Vecinos más próximos	518,98	**0,05**	888,53	**0,05**	1.227,06	0,05	913,47	**0,04**	332.44	**0.01**
Vecinos más próximos repetitivo	453,49	66,22	570,76	67,54	602,57	55,41	558,77	51,36	310.66	6.75
2-Opt	452,95	0,42	570,42	0,63	600,38	0,5	**558,69**	0,28	**299.99**	0.03
Lin-Kernighan	452,97	20,85	570,39	18,58	600,38	22,14	**558,69**	16,97	**299.99**	3.62
Concorde	**452,62**	17,37	**569,77**	20,06	**600,26**	14,11	**558,69**	8,40	**299.99**	2.49
Simulated annealing, 40000 iteraciones	505,91	5,04	657,75	5,08	677,75	5,22	601,86	5,59	340.84	3.65
Simulated annealing, 100000 iteraciones	503,99	12,20	656,83	14,06	664,79	12,56	590,95	11,98	327.98	9.17
Simulated annealing, 1000000 iteraciones	482,97	141,86	611,94	133,66	641,93	120,99	578,84	120,21	309.50	89.64

Tabla 4.2: Comparativa de los algoritmos en longitud de la ruta encontrada (Long) en kilómetros y en tiempo de ejecución (Tmp) en segundos

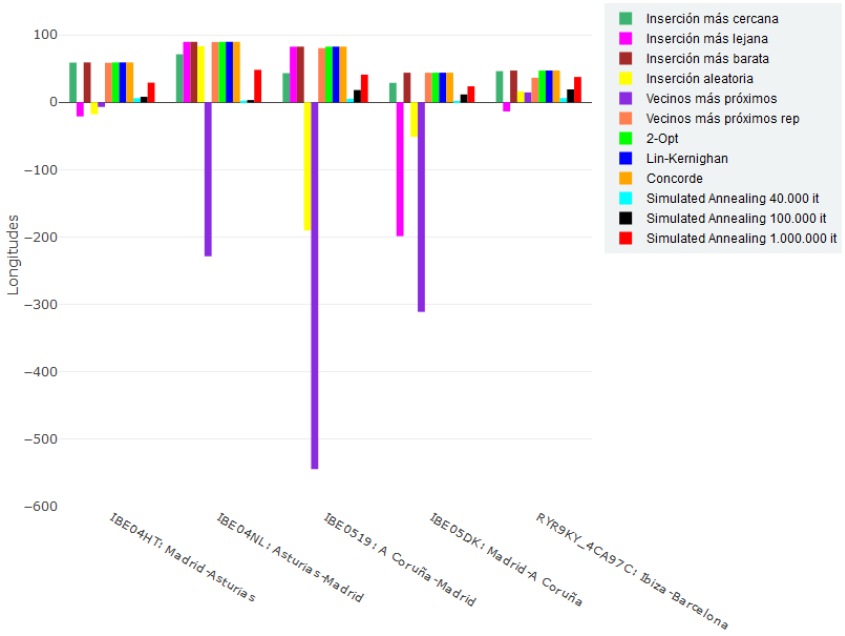


Figura 4.5: Diagrama de barras: Longitud mejorada (en km) registrada con respecto a la ruta de partida inicial

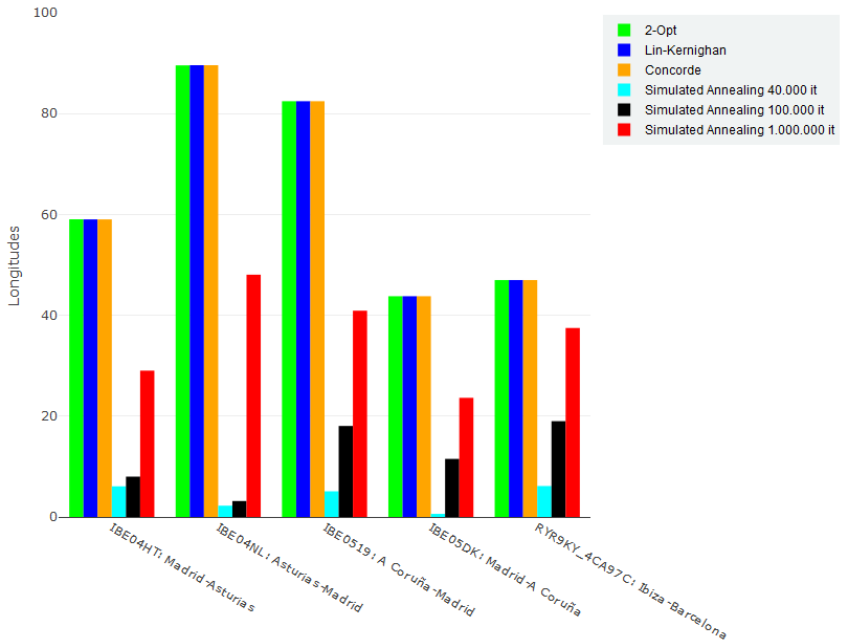


Figura 4.6: Diagrama de barras: Longitud mejorada (en km) registrada con respecto a la ruta de partida inicial obtenida por los métodos no constructivos

En la Tabla 4.2 se muestran los resultados obtenidos. Tal y como se puede apreciar, en los algoritmos constructivos la calidad de las rutas encontradas difiere en función del vuelo. Exceptuamos de esta tendencia irregular al algoritmo de *inserción más barata*, que para todas las trayectorias obtiene rutas muy próximas a las más cortas encontrada por algoritmos de implementación más compleja como *Concorde* o *Lin-Kernighan* (hasta un $0,11\,\%$ más larga que la encontrada por *Concorde* para el IBE04NL).

Por su parte, el algoritmo *2-Opt* obtiene resultados mejores que en el estudio de la Sección 3.2. Las rutas obtenidas por este algoritmo en todos los casos son de longitud muy similar a la mejor encontrada, comportamiento similar al mostrado por el algoritmo de *inserción más barata* (hasta un $0,11\,\%$ más larga que la encontrada por *Concorde* para el IBE04NL). Sin embargo, el tiempo de ejecución requerido es el menor de todos los algoritmos no constructivos. Este nuevo comportamiento del *2-Opt* se explica porque la ruta inicial de la que parte (orden de llegada de los mensajes a estaciones receptoras), es muy similar a la correcta ordenación de los mensajes. En estas condiciones, como se comenta en la Sección 3.1, el *2-Opt* es muy adecuado por el pequeño alcance de las modificaciones que realiza sobre la ruta (denominadas *2-changes*).

En cuanto a *Lin-Kernighan* y *Concorde* se obtienen las mejores rutas, difiriendo levemente, en todos los vuelos. Sin embargo, en tiempo de ejecución es el *Concorde* el que es preferible, requiriendo por ejemplo el $49,5\,\%$ del tiempo necesitado por *Lin-Kernighan* para obtener la misma solución en IBE05DK. A diferencia del estudio de la Sección 3.2, *Concorde* es más eficiente que *Lin-Kernighan*, por lo que resulta preferible en el supuesto de la reordenación de mensajes ADS-B.

Por último, *simulated annealing* mejora levemente la ruta inicial dada, obteniendo resultados cada vez mejores a medida que se aumenta el número de iteraciones consideradas para el algoritmo. Así, para el vuelo IBE04HT, la ruta encontrada es un $1.2\,\%$ más corta que la original para 40.000 iteraciones del algoritmo, un $1.66\,\%$ para 100.000 iteraciones y un $5.67\,\%$ para 1.000.000 iteraciones. Sin embargo, a más iteraciones, más tiempo de ejecución requerido, alcanzando los 2 minutos con 1.000.000 iteraciones en cuatro de las cinco ejecuciones realizadas.

Como conclusión de esta parte, podemos afirmar, por un lado, la eficacia del algoritmo *Concorde* a la cual ahora podemos sumar, a diferencia de lo registrado en la Sección 3.2, los buenos resultados en términos del tiempo de ejecución. Sin embargo, no resulta adecuado para instancias grandes del problema debido principalmente a su complejidad computacional. En estos casos es preferible hacer uso de un algoritmo más simple como el *2-Opt*, cuyos resultados en términos de longitud casi alcanzan los del *Concorde* pero reduciendo en varios órdenes el tiempo de ejecución requerido. Este tiempo es casi inapreciable en la mayoría de los casos para el *2-Opt* frente a los más de 10 segundos que el *Concorde* ha requerido para resolver algunas de las rutas propuestas. El comportamiento del resto de algoritmos, exceptuando al *simulated annealing*, resulta altamente irregular, obteniendo buenos resultados para ciertas instancias y descartables en otras. Por su parte, el *simulated annealing* garantiza mejorar, al menos levemente, la ruta de partida dada, pero requiriendo para ello un elevado tiempo de ejecución.

Capítulo 5

Conclusiones

Este trabajo se ha enfocado a la resolución del *problema de la reordenación de los datos de seguimiento asociados a las trayectorias de aeronaves*, el cual deriva de la problemática surgida al integrar las nuevas tecnologías ADS-B a los sistemas de *gestión del tráfico aéreo* o *Air Traffic Management* (ATM) en el contexto europeo. La solución que ha propuesto este trabajo consiste en modelar el problema como una variante del *problema del viajante* (TSP) en la que las ciudades de origen y de destino son distintas. A su vez, esta consideración obliga a introducir ciertas suposiciones, como la de tomar como reordenación correcta de los mensajes ADS-B aquella que haga mínima la longitud del camino que recorre cada uno de los puntos en orden.

La propuesta de solución que este trabajo aporta ha requerido primeramente de una revisión de los fundamentos teóricos del problema del viajante, en el Capítulo 1, donde se prueba que el problema TSP es fuertemente $\mathcal{NP}-hard$, por el Corolario 1.38, que deriva de uno de los teoremas asociados al matemático estadounidense Stephen Cook (Teorema 1.37).

El modelado del supuesto práctico planteado requiere a su vez de la identificación y el estudio de los algoritmos aplicables para resolver el TSP. Así, en el Capítulo 2 se recogen los más destacados, profundizando sobre las *heurísticas de mejora local*, ya que el estudio de estos métodos, tal y como se citó en la *introducción*, era el segundo de los objetivos de la presente memoria. En el Capítulo 3 se aplicaron estos algoritmos entre otros a la resolución de supuestos generales.

Con la presentación de la propuesta que este trabajo aporta (ver Sección 4.2) y su puesta en práctica sobre trayectorias reales (ver Sección 4.4), se obtienen conclusiones acerca de la adecuación del modelo así como de los diferentes algoritmos de resolución del TSP estudiados a ser aplicados a este supesto.

En esta línea, se puede concluir que el modelo de resolución que este trabajo propone resulta exitoso al asociarlo a ciertos métodos, destacando el *2-Opt* o el *Concorde*.

Por otra parte, la técnica de las *ventanas de ejecución* presentada en la Sección 4.3 para subsanar el alto coste computacional del TSP resulta eficaz al aplicarse junto al algoritmo *2-Opt*.

Finalmente, tras haber sido reordenada una trayectoria, los mensajes ADS-B quedan ordenados de forma inconsistente con su marca temporal o *timestamp*. Un próximo paso a realizar podría consistir en solventar esta incoherencia, construyendo un modelo que, partiendo del nuevo orden de los mensajes tras haber sido reordenados, corrija el *timestamp* de cada uno de ellos a partir del resto de sus datos, de forma que su ordenación ascendente se corresponda con este nuevo orden de los mensajes. Además, las nuevas marcas temporales deberían asemejarse lo más fielmente posible al verdadero instante temporal en que los mensajes ADS-B fueron enviados.

Notación y terminología

$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$	Conjuntos de los números naturales, enteros, racionales y reales respectivamente
$\mathbb{Z}_+$	Conjunto de los números enteros no negativos
$\mathbb{R}^n$	Espacio euclídeo de dimensión n
$A \subsetneq B$	A está contenido estrictamente en B
$A \subseteq B, A \subset B$	A está contenido o es igual que B
$tam(x)$	Longitud de la cadena x
$tiempo(\Phi, x)$	Número de iteraciones que requiere el procesamiento de la cadena x por la máquina de Turing Φ
$salida(\Phi, x)$	Cadena resultante del procesamiento de la cadena x por la máquina de Turing Φ
$x \leq y$	Desigualdad componente a componente, siendo x e y vectores
$\mathcal{A}^n$	Conjunto de las cadenas sobre el alfabeto $\mathcal{A}$ con n elementos
$\mathcal{A}^*$	Conjunto de todas cadenas (finitas) sobre $\mathcal{A}$
$x\#c$	Cadena formada por la concatenación de la cadena x a un certificado asociado c a través del carácter especial de concatenado #
$\mathcal{P}$	Clase de los problemas resolubles polinomialmente
$\mathcal{NP}$	Clase de los problemas resolubles polinomialmente por un algoritmo de comprobación de certificados
$OPT(x)$	Solución óptima de la instancia x de un problema de optimización
mín, máx	Mínimo y máximo
$\|V\|$	Cardinal del conjunto V
$\overline{x}$	Negación lógica de la variable booleana x
$\mathcal{O}(n)$	Notación o-grande
$E(C)$	Conjunto de las aristas que componen el camino C
$x := y$	Operación de asignar el valor de la variable y a la variable x
$Coste(P)$	Coste o longitud del camino P sobre el grafo de una instancia del TSP

Bibliografía

A. Alonso-Isla, P. Álvarez Esteban, A. Bregón, F. Díaz, I. García-Miranda, P. Gordaliza, y M. Martínez-Prieto. Airports: Análisis de eficiencia operacional basado en trayectorias de vuelo. *JISBD*, 2018.

P. Álvarez Esteban, A. Bregón, F. Díaz, I. García-Miranda, y M. Martínez-Prieto. Towards a scalable architecture for flight data management. *DATA 2017 - 6th International Conference on Data Science, Technology and Applications*, pages 263–268, 2017.

G. Babin, S. Deneault, y G. Laporte. Improvements to the or-opt heuristic for the symmetric traveling salesman problem. *The Journal of Operational Research Society. Vol. 58, No. 3*, pages 402–407, 2007.

C. Berge. *Graphs and hypergraphs. Second revised edition.* North-Holland Mathematical Library, 1976.

B. Chandra, H. Karloff, y C. Tovey. New results on the old k-opt algorithm for the traveling salesman problem. *SIAM Journal on Computing*, 1999.

W.J. Cook. *In Pursuit of the Traveling Salesman.* Princeton University Press, 2012.

M. Englert, H. Röglin, y B. Vöcking. Worst case and probabilistic analysis of the 2-opt algorithm for the tsp. *SODA '07 Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1295–1304, 2007.

Eurocontrol. Flight movements and service units 2017-2023. Technical report, Eurocontrol, Febrero 2017. Disponible en :https://www.eurocontrol.int/sites/default/files/content/documents/official-documents/forecasts/seven-year-flights-service-units-forecast-2017-2023-Feb2017.pdf.

Y. Grushka-Cockayne y B. De Reyck. Towards a single european sky. *Interfaces*, 39: 400–414, 2009.

G. Gutin y A. P. Punnen. *The Traveling Salesman Problem and its variations.* Springer, 2007.

D.S. Johnson y L.A. McGeoh. The traveling salesman problem: a case study. En E. Aarts y J. K. Lenstra, editors, *Local Search in combinatorial optimization*, chapter 6. John Wiley & Sons Ltd, 1997.

B. Korte y J. Vygen. *Combinatorial Optimization, Theory and Algorithms*. Springer, 2006.

G. Laporte. The traveling salesman problem: An overview of exact and approximated algorithms. *European Journal of Operational Research*, 59:231–247, 1992.

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, y D.B. Shmoys. *The Traveling Salesman Problem*. Wiley, 1992.