Universidad de Cantabria

Facultad
de
Ciencias

OPTIMIZACIÓN DEL PROBLEMA DEL VIAJANTE: MÉTODOS DE BRANCH AND BOUND Y LIN KERNIGHAN

(Optimisation of the Travelling Salesman Problem: Branch and Bound and Lin-Kernighan Methods)

Trabajo de Fin de Grado
para acceder al

GRADO EN MATEMÁTICAS

Autora: María Pino Rey

Directora: Mónica Blanco Gómez

Septiembre - 2024

Resumen

El problema del Viajante (TSP) es formulado como un problema de programación lineal entera que trata de encontrar un ciclo hamiltoniano que minimice los costes de las aristas utilizadas sobre un grafo completo ponderado $G = (V, A)$.

Al tratarse de un problema NP-completo tiene diversas técnicas de resolución tanto exactas como heurísticas. Se planteará el método exacto de Branch and Bound con la regla de ramificación Little y el método heurístico de mejora iterativa $k-opt$ Lin Kernighan, además de exponer pequeños problemas ilustrativos de ejemplo para ver el procedimiento de los métodos.

Finalmente se implementarán en Matlab los algoritmos mencionados para la resolución de dos problemas reales comparando las soluciones y tiempos de ejecución.

Palabras clave: Problema del viajante, grafo, programación lineal, ramificación, intercambio

Abstract

The Traveling Salesman Problem (TSP) is formulated as an integer linear programming problem what tries to find a Hamiltonian cycle that minimises the costs of the edges used on a weighted graph $G = (V, A)$.

As it is an NP-complete problem, it has several exact and heuristic solution techniques. The Branch and Bound exact method with the Little branching rule and the Lin Kernighan heuristic iterative improvement method will be presented, as well as small illustrative example problems to show the procedure of the methods.

Finally, the aforementioned algorithms will be implemented in Matlab to solve two real problems, comparing the solutions and execution times.

Key words: Travelling Salesman Problem, graph , linear programming, branching, exchange

Agradecimientos

Agradecer a mi tutora, Mónica, por aceptar guiarme en este proyecto, ayudarme cuando lo necesité, toda su dedicación y todo su tiempo empleado.

A todos mis profesores, si no fuera por ellos no sería posible la realización de este trabajo.

A mis amigos, los que ya lo eran y los que tuve la suerte de que ahora lo sean. Por estar siempre ahí y confíar en mí incluso cuando ni yo misma lo hacía, celebrando desde la distancia mis logros como si fueran los suyos.
A Sergi, por hacerme sin saberlo, los malos momentos mucho más llevaderos.

A mi familia, que me dieron ánimo y fuerza para que nunca me diese por vencida. En especial a mi tía, por estar siempre más que pendiente de su "tercera hija".

A mis abuelos, que seguro que estarán muy orgullosos de mi.
¡Abu, lo conseguí!

A mi madre, mi ejemplo a seguir, por ser mi gran apoyo incondicional y enseñarme que con trabajo y esfuerzo conseguiré todo lo que me proponga.

Y por último como todo lo que hago, va dedicado a ti, Papá, por no dejar que nunca me rinda y acompañarme siempre en cada paso que soy capaz de dar.

Índice general

Capítulo 1

Introducción

Este documento estudiará el Problema del Viajante, uno de los problemas más estudiados en el campo de las Matemáticas, desde las perspectivas de la Teoría de Grafos y la Programación Lineal Entera abordando su resolución mediante dos técnicas específicas explicadas tras la recopilación de diversas fuentes y autores. La motivación principal es desarollar una aplicación práctica que demuestre su utilidad en situaciones cotidianas a través de una de sus múltiples aplicaciones como es la logística o el transporte.

La estructura de este documento es la siguiente:

En el este capítulo se expondrá una serie de preliminares para comprender de manera más clara el contexto de lo que se tratará a continuación.

En el segundo capítulo, se planteará el problema más en profundidad así como la explicación de sus orígenes y la formulación que será empleada en todo el documento.

El tercer capítulo, tratará de explicar el método exacto Branch and Bound ilustrandolo con un breve ejemplo práctico además de la explicación de su implementación en *Matlab*. El cuarto, tomará la misma estructura que el capítulo anterior pero en este caso será explicado el método heurístico Lin Kernighan.

Por último, se desarrolla una de sus aplicaciones más conocida, el ámbito de la logística/transporte. Es planteada desde dos puntos de vista diferentes; teniendo en cuenta la propia red de carreteras existente en España, concretamente en Galicia, y creando de manera artificial carreteras que conectan de manera rectilínea todos los puntos escogidos del mapa. Para el análisis de los resultados por los diferentes métodos planteatos en el documento como se dijo, se implementaron los métodos en el programa *Matlab*.

Se darán una serie de definiciones que serán de utilidad para la comprensión del documento.[7, 5, 6, 4, 20, 18]

1.1. Preliminares de Teoría de Grafos

Definición 1.1.1. *Un **grafo (finito) simple** es una terna $G = (V, A, \varphi)$ donde $V = V(G)$ es un conjunto finito no vacío (cuyos elementos se denominan vértices o nodos), $A = A(G)$ es un conjunto cuyos elementos se llaman aristas y $\varphi : A \longrightarrow P(V)$ es una aplicación que asocia a cada arista un par de vértices. Es decir $\varphi(x) = \{i, j\}$ con $x \in A$ y $i, j \in V$. Se dice **simple** si no existen lazos ($i \neq j \ \forall \varphi(x) = \{i, j\}$) y dos vértices están unidos a lo sumo por una arista.*

Definición 1.1.2. *Un grafo es **completo**,denotado por K_n, si $A = V \times V$,es decir, para cualquier pareja de vértices, existe una arista que los une. Siendo $\mid A \mid = \mid V \mid^2$.*

Definición 1.1.3. *Un **grafo dirigido o digrafo** es una terna $G = (V, A, \varphi)$ donde V, A son como en la definición de grafo y $\varphi : A \longrightarrow V \times V$ es una aplicación que asocia a cada arista un par de vértices ordenados (una arista empieza en x y acaba en y).*

Definición 1.1.4. *Un grafo dirigido es **simétrico** si para toda arista $(i, j) \in A$ también se cumple que $(j, i) \in A$.*

Definición 1.1.5. *Un **camino** es una sucesión $i_0 x_1 i_1 x_2 i_2 \cdots i_{k-1} x_k i_k$ donde $i_s \in V$ y $x_s \in A$ una arista incidente a i_{s-1} y i_s donde todas las aristas son distintas y $s \in \{1, \ldots k\}$.Se dice **cerrado** si $i_0 = i_k$ con $i_s \in V$. Se dice **simple** si no hay vértices repetidos salvo el primero y el último.*

Definición 1.1.6. *Un **circuito** es un camino cerrado.*

Definición 1.1.7. *Un **ciclo** es un camino cerrado simple no trivial (no es lazo).*

Definición 1.1.8. *Un **grafo pesado o ponderado** en aristas es un grafo $G = (V, A)$ junto con una aplicación $w : A \longrightarrow \mathbb{R}_+$ que asigna a cada arista un coste. El **peso de un camino** es la suma de los pesos de las aristas que la componen.*

Definición 1.1.9. *Dados dos vértices i, j en un grafo pesado se llama **distancia** entre i e j, $d(i, j)$, al mínimo de los pesos de caminos que unen i con j.Un camino $i_0 e_1 i_1 \cdots x_k i_k$ es de **peso mínimo** si su peso es $d(i_0, i_k)$.*

Definición 1.1.10. *Un **subgrafo** de un grafo $G = (V, A)$ es un grafo $H = (W, F)$ donde $W \subseteq V$ y $F \subseteq A$*

Definición 1.1.11. *Sea $G = (V, A)$ un grafo simple .El **subgrafo inducido** por un subconjunto $W \subset V$, denotado por $G[W]$, es el grafo $H = (W, F)$ donde el conjunto de aristas F contiene una arista en A si y solo si ambos extremos de esta arista están en W. Es decir, si el subgrafo H contiene todas las aristas que unen en G dos vértices de W.*

Definición 1.1.12. *Un **árbol** es un grafo conexo que no contiene ciclos y tiene $n - 1$ aristas.*

Definición 1.1.13. *Sea G un grafo y $H \subseteq G$ un subgrafo. Se dice que H un **árbol generador** de G si H es un árbol con $V(H) = V(G)$.*

Definición 1.1.14. *Un **árbol dirigido** es un árbol que tiene un vértice distinguido al que se le llama **raíz** . Siendo a_0 el nodo raíz, se le llamará a a_1 padre de a_2 y a_2 hijo de a_1 .*

Definición 1.1.15. *Un **árbol binario** es aquel árbol que tiene exactamente dos hijos.*

Definición 1.1.16. *Un algoritmo de **búsqueda de anchura** sobre un árbol trata de construir un árbol a partir de un vértice inicial i añadiendo los vértices sucesivos según su ditancia al vértice original.*

Definición 1.1.17. *Un algoritmo de **búsqueda en profundidad** trata de construir un árbol a partir de un vértice inicial i intentando volver atrás lo menos posible.*

1.2. Preliminares de Programación lineal

Definición 1.2.1. *Un **Problema de Optimización (P)** de dimensión finita puede definirse como:*
Dada $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ y $K \subseteq \mathbb{R}^n$, el problema consistente en hallar $\overline{x} \in K$ tal que $f(\overline{x}) \leq f(x) \forall x \in K$ formulado en la forma

$$(P) \begin{cases} \text{Minimizar} & f(x) \\ \text{sujeto a} & x \in K \end{cases}$$

*Se formula a partir de un conjunto de variables independientes,x, sobre las que pueden existir condiciones ($x \in K$), llamadas **restricciones del problema** y una función dependiendo de las variables que se quiere minimizar llamada **función objetivo**.*

Definición 1.2.2. *Un problema de optimización se considera un problema de **Programación Lineal(LP)** si f es lineal, es decir, $f(x) = c^T x$ con $c \in \mathbb{R}^n$ y escrito es su forma estándar, $K = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$ siendo $A \in \mathbb{R}^{m \times n}$ y $b \in \mathbb{R}^m$.*

Definición 1.2.3. *Un problema de **Programación Lineal Entera Mixta(MILP)** es un problema lineal (LP) con algunas variables enteras.*

Definición 1.2.4. *Un problema de **Programación Lineal Entera 0-1** es un problema MILP en donde las variables unicamente toman valores 0 y 1.*

Definición 1.2.5. *Un problema lineal se dice **relajado** si se prescinde de alguna restricción del problema original.*

Definición 1.2.6. *Un punto x que satisface todas las restricciones del problema, se considera **solución factible**. El conjunto de todas las soluciones factibles se le llama **región factible**.*

Definición 1.2.7. *Una **solución óptima** ,$\overline{x}$ es el punto o puntos de la región factible que minimiza la función objetivo, es decir, cuando se cumple que $f(\overline{x}) \leq f(x)$ para toda x solución factible.*

Definición 1.2.8. *En un problema lineal 0-1 se le dice **solución parcial** a la solución en donde algún valor está todavía sin fijar.*

Definición 1.2.9. *Se dice que $\overline{x}$ es un **mínimo global** de P si verfica $f(\overline{x}) \leq f(x) \forall x \in K$*

Definición 1.2.10. *Se dice que $\overline{x}$ es un **mínimo local** de P si existe un entorno abierto de $\overline{x}$, U, tal que se verifica $f(\overline{x}) \leq f(x) \forall x \in K \cap U$*

Capítulo 2

El problema

2.1. Planteamiento

El Problema del Viajante o también conocido como TSP por sus siglas del inglés Travelling Salesman Problem, es un problema de optimización combinatoria cuyo número de posibles soluciones crece exponencialmente con el tamaño del input. El problema,responde a la siguiente pregunta:

"Dada una lista de n ciudades y las distancias entre cada par de ciudades, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y vuelve a la ciudad de origen?".

El TSP puede plantearse como un problema de Teoría de Grafos. Los nodos se corresponden con las ciudades, las aristas con los caminos entre ciudades y los pesos de las aristas con los valores de las distancias de ir de un lugar i a otro j representados como c_{ij} .

Partiendo de n lugares como una red de nodos, para cada arista (i, j) se tiene un valor c_{ij} (Definición 1.1.8), tratando de encontrar en qué orden recorrer los nodos de la red de modo que minimice la distancia total recorrida; dicho de otro modo, encontrar un circuito hamiltoniano (Definición **??**) con peso mínimo donde es indiferente el lugar que se designe como origen.

A su vez, puede ser planteado desde la Programación Lineal satisfaciendo lo que se denomina función objetivo. Concretamente, atendiendo a la linealidad y naturaleza de las variables, se habla de Programación Lineal Entera 0-1 (Definición 1.2.1) por tomar las variables x_{ij}, que se describirán más adelante, valores 0 y 1.
Dentro de la Programación Lineal Entera, se encuentra la llamada Optimización Combinatoria, donde hay un número finito aunque elevado de posibles decisiones. El TSP forma parte de esta subclase de problemas.

Según su simetría puede clasificarse en sTSP o TSP simétrico, donde es indiferente en qué sentido recorrer las aristas, ya que tienen igual peso en ambos y en aTSP o TSP asimético donde sí importa el sentido de recorrido. De manera práctica, si una carretera es de doble sentido se hablará de sTSP y si por la contra es de sentido único aTSP.

En general se hablará de TSP, ya que el sTSP es un caso particular del asimétrico.

2.2. Historia

El origen de este problema no está del todo claro, aunque se cree que se puede fechar por una guía para vendendores ambulantes que incluye viajes de ejemplo por Alemania y Suiza, en el 1832. Una versión de resolución mediante la Teoría de Grafos se puede datar en 1736 cuando Leonhard

Euler resolvió el Problema de los Siete Puentes de Königsberg o el Problema del caballo de ajedrez.

El primero de los problemas, el nombre, se debe a Königsberg, una ciudad de Prusia Oriental y más tarde de Alemania que desde 1945 se convirtió en la ciudad rusa de Kaliningrado. Esta ciudad está atravesada por el río Pregolia que se bifurca y rodea con sus brazos a la isla Kneiphof quedando así dividido en cuatro regiones distintas que entonces estaban unidas mediante siete puentes.[21]

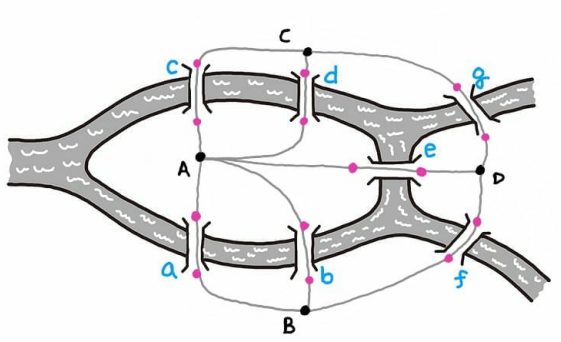

Figura 2.1: Grafo sobre mapa con los puentes de Königsberg [13]

Como se dijo, Euler resolvió el problema representando como se puede ver en la Figura 2.1 a la ciudad de Königsberg en una especie de grafo donde los puentes a, b, c, d, e, f, g delimitados por dos puntos (en rosa) determinan las cuatro partes del mapa $\{A, B, C, D\}$ que se corresponderán con los vértices. El objetivo era encontrar un recorrido para cruzar a pie toda la ciudad, pasando solo una vez por cada uno de los puentes regresando al mismo punto de inicio. Años más tarde, él mismo, concluyó que era imposible.

En el segundo de los problemas, se pedía que teniendo una cuadrícula de $n \times n$ casillas y un caballo de ajedrez colocado en una posición cualquiera (a, b), el caballo pasase por todas las casillas una sola vez con su movimiento en L y regresase a su punto de partida.
Este problema fue resuelto con éxito para el caso de un tablero de ajedrez completo (8×8) por diversos expertos y métodos. Sin embargo, para otras dimensiones de tablero no es posible obtener un resultado positivo. Es el caso de tableros en los que el número de casillas es impar o los tableros rectangulares o en forma de cruz. Véase la demostración de la primera afirmación:

Ejemplo 2.2.1. *Un tablero de ajedrez con un número impar de casillas no puede ser recorrido por el caballo sin repetir ninguna casilla y volviendo al punto de partida.*

Se resolverá atendiendo a la coloración de las casillas del tablero (blanco o negro). Supongamos sin perder generalidad que el caballo comienza en una casilla blanca. Su movimiento en L alterna tras cada movimiento el color de la casilla. Como se requiere que el punto final sea el de partida según la suposición, se debe terminar en una casilla blanca. Esto es imposible de conseguir, ya que necesitaría realizar un número par de movimientos, pero el número de casillas es impar.

En 1857, se desarrolló el también conocido Icosian Game por Willian Rowan Hamilton que consistía en un desafío matemático cuyo objetivo irradiaba en dar con el recorrido hamiltoniano por las aristas de un dodecaedro para visitar una y solo una vez cada vértice coincidiendo el de llegada con el de salida.
En la figura siguiente se representa una posible solución:

Desde otras perspectivas de resolución, en el siglo XIX el TSP fue formulado matemáticamente por William Rowan Hamilton y Thomas Kirkman y desde entonces, ha sido estudiado por diversos expertos en la materia. Tomó gran popularidad entre los años 50 y 60 debido a su estrecha relación con los problemas de programación lineal de asignación y transporte. Así, en 1954 “Soluciones

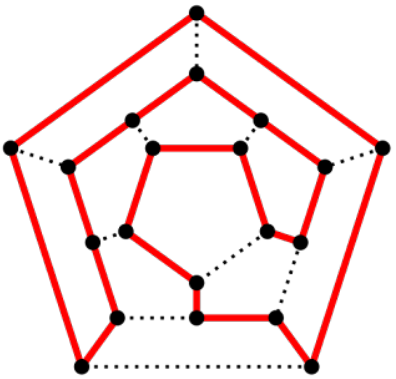

Figura 2.2: Posible solución al Icosian Game

de un problema del viajante de gran tamaño"de Dantzig, Fulkerson y Johnson en el Journal of the Operations Research Society of America fue uno de los principales eventos en la historia de la optimización[8]. El método utilizado parecía funcionar en pequeños problemas con unas 49 ciudades.

A continuación se muestra una tabla que contiene los avances en la resolución del TSP según el número de nodos/ciudades a tener en cuenta [19]:

Año	Autores	Nodos
1954	G. Dantzig, R. Fulkerson, S. Johnson	49
1971	M. Held, R. Karp	64
1977	M. Grötschel, M. Padberg	120
1980	H. Crowder, M. Padberg	318
1987	M. Padberg, G. Rinaldi	532
1987	M. Grötschel, O. Holland	666
1987	M. Padberg, G. Rinaldi	2.392
1992	V. Chvátal, W. Cook (Concorde)	3.038
1994	D. Applegate, R. Bixby, V. Chvátal, W.Cook	7.397
1998	D. Applegate, R. Bixby, V. Chvátal, W. Cook (Concorde)	13.509
2001	D. Applegate, R. Bixby, V. Chvátal, W. Cook, K. Helsgaun (Concorde)	15.112
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, K. Helsgaun (Concorde)	24.978
2006	V. Chvátal, W. Cook (Concorde)	85.900
2009	Y. Nagata	1.000.000
2010	S. Lin, B. Kernighan, K. Helsgaun	1.904.711

Tabla 2.1: Evolución del TSP según el número de ciudades a considerar.

* Concorde es un código informático en lenguaje de programación C para resolver el problema del viajante simétrico u otros problemas de redes relacionados.

2.3. Formulación TSP

En la sección anterior, se expuso la primera formulación del TSP sobre un grafo (Figura 2.2) tratando de encontrar un ciclo hamiltoniano, pero además, el problema puede ser formulado como un problema de Programación Lineal Entera, como se verá a continuación.

El TSP puede ser formulado como un grafo completo no dirigido(sTSP) o dirigido (aTSP) en donde las aristas $A = \{(i,j)\ \forall i,j \in V, i \neq j\}$ tienen distintos pesos o costes. Teniendo en cuenta la Definición 1.1.8 puede construirse la matriz de costes $C = [c_{ij}]$ de dimensión $n \times n$, por ser n

las ciudades distintas que existen, donde cada c_{ij} representa el peso de la arista (i, j). En forma matricial se puede ver como:

$$C = \begin{bmatrix} \infty & c_{12} & \dots & c_{1n} \\ c_{21} & \infty & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & \infty \end{bmatrix}$$

En el caso del sTPS se cumplirá que $c_{ij} = c_{ji}$ por tratarse de un grafo no dirigido. Los elementos c_{ij} siempre tomarán valores positivos y en caso de que dos vértices no tengan una arista que los conecte, c_{ij} tomará un valor arbitrariamente grande para que el algoritmo no tenga en cuenta dicha arista. Cuando el grafo es completo, los únicos elementos que tomarán "valor ∞ " serán los de la diagonal, ya que no es posible ir de una ciudad en sí misma.
Teniendo un grafo, la construcción de la matriz de costes se realiza de manera muy sencilla.

Ejemplo 2.3.1. *Supongamos que se tiene el grafo de 6 vértices que se muestra a continuación:*

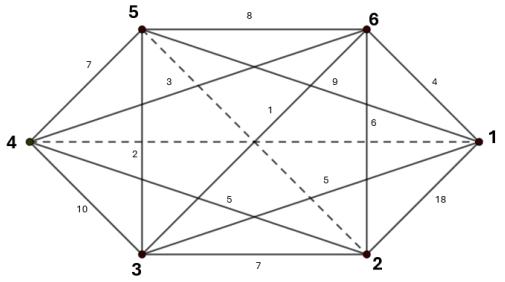

Figura 2.3: Grafo ponderado de 6 vértices

Las aristas punteadas no existían en el grafo inicial, por lo que se incluyen de manera artificial con "coste infinito"para la creación de la matriz de costes asociada:

$$C = \begin{bmatrix} \infty & 18 & 5 & \infty & 9 & 4 \\ 18 & \infty & 7 & 5 & \infty & 6 \\ 5 & 7 & \infty & 10 & 2 & 1 \\ \infty & 5 & 10 & \infty & 7 & 3 \\ 9 & \infty & 2 & 7 & \infty & 8 \\ 4 & 6 & 1 & 3 & 8 & \infty \end{bmatrix}$$

Por otra parte, formulándolo desde la Programación Lineal Entera, se plantea el problema de la siguiente manera:

- Se enumeran las ciudades de $1, \dots, n$. Para cada par de ciudades (i, j) se consideran x_{ij}, variables booleanas, de modo que tomarán el valor 1 si la arista (i, j) está en la solución y 0 en otro caso.
- Sea c_{ij} la distancia desde la ciudad i a la ciudad j.
- El objetivo del problema y, por tanto, función a optimizar es conseguir, como se vino diciendo hasta ahora, un ciclo Hamiltoniano de peso mínimo, por tanto, restricciones a tener en cuenta serán que cada vértice tenga tan solo una arista de salida y una de entrada.

Con las variables, restricciones y objetivo descritos se construye el modelo general de Programación Lineal en Enteros siguiente:

$$(P_0) \quad \text{Minimizar } f= \sum_{i,j\in V} c_{ij}x_{ij} \tag{2.1}$$

sujeto a

$$\sum_{j\in V} x_{ij} = 1 \quad \forall i \in V \setminus \{j\} \tag{2.2}$$

$$\sum_{i\in V} x_{ij} = 1 \quad \forall j \in V \setminus \{i\} \tag{2.3}$$

$$x_{ij} = \{0,1\} \quad \forall i,j \in V \tag{2.4}$$

La ecuación (1) será la función objetivo, mientras que la (2) y la (3) serán las restricciones que impondrán que se salga y se llegue una sola vez a cada ciudad respectivamente.

Notación 1. *Dada una solución factible T del problema P_0 se puede representar por el conjunto de aristas que forman el ciclo, $T = [(i_1,i_2),(i_2,i_3),\ldots,(i_{n-1},i_n)(i_n,i_1)]$ .El coste total de dicho ciclo será $z(T) := \sum_{(i,j)\in T} c_{ij}$. Además, si T es una solución óptima, $z(T)$ tendrá el menor coste posible y será, por tanto, una cota superior de $v(P)$ (valor óptimo de P).*

Observación 2.3.2. *Por como se define $z(T)$, se escoge uno y solo un elemento (coste) en cada fila y cada columna de la matriz C para contrubuír al coste final de la solución.*
Por otro lado, para el caso simétrico, la solución $\overline{T} = [(i_n,i_{n-1}),(i_{n-2},i_{n-3}),\ldots,(i_2,i_1)(i_1,i_n)]$ tendrá coste, $z(\overline{T}) = z(T)$ aunque se traten de dos soluciones distintas.
Para ambos casos (simétrico o asimétrico) si $x_{ij} = 1$ automáticamente $x_{ji} = 0$ y viceversa

Aunque estas restricciones son necesarias, no son suficientes ya que se podrán formar ciclos disjuntos como ocurre en esta posible solución de (P_0) sobre la Figura 2.3:

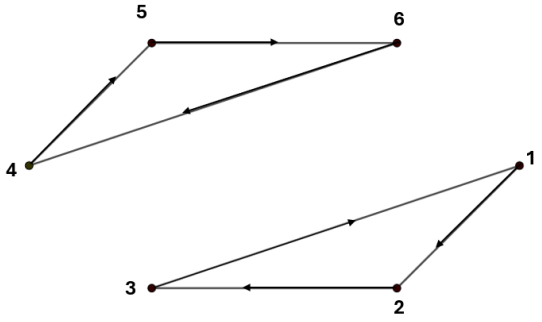

Figura 2.4: Ciclos disjuntos con 6 nodos

En la Figura anterior, $x_{12} = x_{23} = x_{31} = x_{45} = x_{56} = x_{64} = 1$ por lo que se cumplen las restricciones de (P_0), sin embargo, no es un circuito hamiltoniano y, por tanto, no es una solución factible para la resolución del problema que se quiere plantear. La ruptura de estos subcircuitos puede ser abordada de varias formas introduciendo distintas restricciones.

Para cada subconjunto $S \subseteq V$ con $2 \leq |S| \leq |V| - 1$, una condición suficiente para evitar subciclos entre los vértices de S es la propuesta por Dantzig, Fulkerson y Johnson (DFJ):

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

A continuación se demuestra que esta restricción sería suficiente para la eliminación de los ciclos de menor longitud.

Proposición. 2.3.3. *Dado un subconjunto $S \subseteq V$ y una solución factible T del problema P_0. Si $\sum_{i,j \in S} x_{ij} \geq |S| \Longrightarrow$ la solución T contiene al menos un ciclo entre los vértices de S .*

Observación 2.3.4. *Si $|V(G)| = n$ y $|A(G)| > n - 1 \Longrightarrow G$ tiene al menos un ciclo.*

Demostración de Proposición 2.3.3. Sea G el grafo, $G[S]$ el subgrafo inducido por los vértices de S (Definición 1.1.11) y sea $T_S = T \cap A(G[S])$. La condición $\sum_{i,j \in S} x_{ij} \geq |S| \Longrightarrow T_S$ tiene al menos $|S|$ aristas.
Por la Observación 2.3.4 como T_S tiene más de $|S| - 1$ aristas $\Longrightarrow T_S$ tiene al menos un ciclo.

□

Añadiendo esta restricción, el problema queda planteado como :

Problema 1 (Formulación del TSP)**.**

$$(P) \quad \textit{Minimizar } f = \sum_{i,j \in V} c_{ij} x_{ij} \tag{2.1}$$

sujeto a

$$\sum_{j \in V} x_{ij} = 1 \quad \forall i \in V \setminus \{j\} \tag{2.2}$$

$$\sum_{i \in V} x_{ij} = 1 \quad \forall j \in V \setminus \{i\} \tag{2.3}$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1 \quad (S \subset V, 2 \leq |S| \leq |V| - 1) \tag{2.4}$$

$$x_{ij} = \{0, 1\} \quad \forall i, j \in V \tag{2.5}$$

Observación 2.3.5. *Los métodos de resolución que se exponen en las secciones sucesivas, serán planteados basándose en esta formulación.*

Definición 2.3.6. *A lo largo del trabajo, a veces se llamará **tour** a una solución factible T y se llamará **cadena** a los camino disjuntos más largos que forman parte de la solución parcial.*

La mayoría de los documentos sobre este problema, hablan sobre su carácter *NP-completo* que hace que esté en cotinuo estudio hasta la obtención de un método lo suficientemente eficiente. Para entender este concepto es necesario introducir la teoría de complejidad.

2.4. Complejidad Temporal

La Teoría de la Complejidad, estudia la forma de clasificar problemas algorítmicos según la dificultad para resolverlos. La clasificación tiene principalmente dos factores en cuenta: el tiempo que tarda el algoritmo en ejecutarse y la memoria necesaria para almacenar los datos tras la ejecución del problema.[23, 2, 24]

Definición 2.4.1. *Se denomina función de complejidad a $T : \mathbb{N} \longrightarrow \mathbb{N}$ donde $T(n)$ es el tiempo transcurrido en la ejecución del algoritmo. Un problema resuelto en **tiempo polinómico** equivale a decir que dado un $n \in \mathbb{N}$, $T(n)$ está acotado por una función polinómica.*

Para poder clasificar un problema según su complejidad algorítmica, debe estar planteado como un problema de decisión, es decir donde las únicas respuestas posibles sean "Sí" o "No".
El TSP planteado como un problema de decisión:

"¿Dado un número $p \in \mathbb{N}$, existe un circuito Hamiltoniano de longitud menor o igual a p?"

La **complejidad temporal** puede ser usada para describir la cantidad de pasos necesarios para resolver un problema o para describir cuanto tiempo lleva verificar si una solución es o no la óptima.

Definición 2.4.2. *La **clase P** está formada por el conjunto de los problemas que pueden ser resueltos con algoritmos en tiempo polinómico explorando una a una las distintas posibilidades.*

Definición 2.4.3. *La **clase NP** abarca el conjunto de todos los problemas de desición que pueden ser resueltos con algoritmos de tiempo polinómico explorando todas las posibilidades a la vez hasta encontrar la correcta dicho de otro modo, si la respuesta al problema es "Sí" puede ser confirmado en tiempo polinomial.*

A día de hoy, aún no se sabe si $P = NP$. Es uno de los problemas abiertos que el Instituto Clay expone en los problemas del milenio en [3]. A raíz de las distintas posibles soluciones que recibió este problema se define:

Definición 2.4.4. *Un problema es **NP-completo** si es NP y todo problema NP se puede reducir*[1] *en tiempo polinomial en NP-completo.*

Por lo que el TSP según su complejidad algorítmica temporal, Richard M. Karp en 1972 lo definió como NP-completo.

Al categorizarse en este grupo, es complicado obtener la solución exacta en un tiempo computacionalmente bajo. Se utilizan métodos que pretenden obtener una solución óptima en el menor tiempo posible.
Un breve resumen de la clasificación general de los métodos es la que se muestra a continuación:

2.5. Técnicas de Resolución

- **Algoritmos Exactos**: Métodos que buscan obtener la solución óptima. Suelen aplicarse a problemas sencillos ya que el tiempo de ejecución es muy elevado.
- **Algoritmos Heurísticos**: Métodos que tratan de encontrar una *buena* aunque no siempre óptima solución al problema con bajos tiempos de ejecución. Son empleados en problemas donde la rapidez y la calidad de la solución priman de igual manera. Suelen ser algoritmos *ad hoc* o lo que es lo mismo, se diseñan para un tipo de problema en específico por lo que consiguen acercarse a la solución en un tiempo razonable. Son utilizados cuando no se conoce ningún algoritmo exacto que pueda resolver el problema o porque usandolo, es demasiado costoso computacionalmente. A lo largo del documento, se empleará la palabra heurística para referirse a estos métodos.
  Una breve clasificación dentro de este grupo es la siguiente:

  - Constructivos: Construyen paso a paso una solución factible añadiendo en cada iteración un nuevo elemento. Algunos de los más conocidos son:
    - Heurísticos Voraces: Eligen en cada paso el mejor mínimo local. Toman decisiones que parecen ser óptimas en el momento pero sin considerar las posibles consecuencias a largo plazo.

[1] Un problema de decisión R es polinomialmente reducible a un problema de decisión Q si hay una transformación en tiempo polinomial de cada ejemplo I_R del problema R a un ejemplo I_Q del problema Q, donde los ejemplos de I_R y I_Q tienen la misma respuesta

- Estrategia de descomposición: Divide el problema inicial en subproblemas más sencillos de resolver individualmente. La solución final del problema original es la combinación de las soluciones de los subproblemas.

- Métodos de búsqueda : Parten de una solución factible y la intentan mejorar proguesivamente obteniendo en cada paso una nueva solución.
  - Heurísticos de mejora iterativa: Utilizando una solución inicial se extraen k aristas con el objetivo de añadirlas de modo que mejore la solución de partida.
  - Heurísticos de multiarranque: Con ayuda de probabilidades y procesos aleatorios mejoran la solución de partida.

El algortimo voraz más conocido e importante es el del *Vecino más próximo* que comienza en una ciudad arbitraria y mientras haya ciudades no visitadas, se visita la ciudad más cercana que todavía no haya aparecido en el recorrido (volviendo por último a la primera ciudad).

2.5.1. Vecino más próximo

Este método construtivo voraz trata de crear un ciclo Hamiltoniano de bajo coste. Dado un vértice, se busca la arista con menor peso que sale de dicho vértice. Las primeras aristas escogidas que aparentemente tienen poco peso pueden causar malas elecciones al final del proceso ya que sólo quedarán aristas con peso muy elevado.

Algorithm 1 Algoritmo Vecino más cercano

INPUT: Matriz costes C

1. *k=0* Seleccionar un vértice $j \in V$ al azar para inicializar $v_0 = j$ y sea $W = V$.
2. *k=k+1* Buscar el vértice más cercano a v_k en $W = V \setminus \{\text{vértices ya visitados}\}$. Si el ciclo está formado por los vértices $v_1, v_2, \ldots, v_k$, se busca el vértice que pasará a formar parte del ciclo v_{k+1} tal que $c_{v_k v_{k+1}} = min\{c_{v_k i} \text{ con } i \in W\}$.
3. Se sigue el proceso hasta que $W = \emptyset$. Una vez finalizado, v_n será unido a v_1 (considerando v_1 el primer vértice y v_n el último).

A continuación, para ver de manera gráfica este método, se representa un ejemplo con un grafo de 5 vértices (partiendo del vértice A) en el que aparece resaltada la solución final:

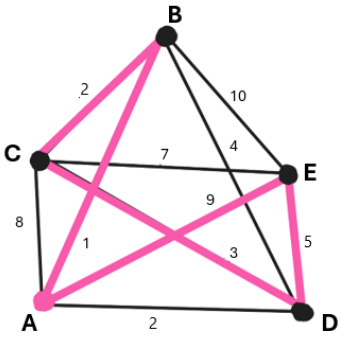

Figura 2.5: Ciclo Hamiltoniano en grafo de 5 vértices creado mediante el método Vecino más cercano

El coste total del recorrido por este método será $1 + 2 + 3 + 5 + 9 = 20$. Una mejor elección del vértice E habría sido la arista que sale hacia A con peso 2, sin embargo ese vértice ya formaba parte del recorrido por lo que se tuvo que añadir una arista de mayor coste.
La solución óptima al problema, sería el ciclo $[ABCED]$ cuyo coste es $1 + 2 + 7 + 5 + 2 = 17$.

El auge en la resolución de este problema mediante diversas técnicas se debe a su importancia práctica en la planificación de rutas, la secuenciación del genoma o la producción de chips entre otros. Encontrar soluciones óptimas y eficientes puede tener un gran impacto en la eficiencia operativa. Por lo tanto, la gran variedad de métodos desarrollados refleja la necesidad de abordar este problema desde múltiples enfoques para satisfacer las necesidades específicas de las distintas aplicaciones.

Capítulo 3

Ramificación y acotación *(Branch and Bound)*

Los métodos exactos garantizan encontrar la solución en un número acotado de pasos. La manera más intuitiva de resolver un problema de tales características es calcular las $n!$ posibles rutas con sus costes asociados y concluir cuál tiene menor coste, es decir, mediante la *búsqueda por fuerza bruta*. Sin embargo, para $n = 9$ ya se calcularían $9! = 362880$ rutas, por lo que tendría un coste computacional demasiado elevado. Es por esto que se proponen técnicas más rápidas y menos costosas como es el algoritmo de ramificación y acotación [25].

El término ramificación y acotación fue introducido en 1963 por John D. C. Little, Katta G. Murty, Sweeney y Karel [16]. Como indica su nombre, el algoritmo consta de la realización de dos pasos.

- Primeramente, el método consiste en resolver el problema de optimización diviendo la región factible en subconjuntos más pequeños determinados por soluciones parciales y representados como nodos de un árbol de decisión. El proceso de división de los subconjuntos (de cada uno de los nodos) radica en la ampliación de la solución parcial fijando una variable no fijada. Esto es lo que se llama ramificación del nodo, pudiendo subdividirse cada uno en al menos dos subconjuntos no vacíos, puesto que cada variable toma dos posibles valores 0 o 1.

- En este proceso de ramificación se van calculando cotas inferiores y superiores del valor de $v(P)$ en cada uno de los subproblemas, asociadas al coste de la solución parcial que corresponda. Esto permite en última instancia decidir si una solución factible es óptima.

El algoritmo puede ser empleado para la resolución de problemas pequeños que contienen entre 40 y 60 ciudades. [12]

Este método puede plantearse de maneras muy diversas dependiendo de la relajación aplicada sobre el problema (P) o la regla de ramificación que determinará el orden de selección de los subproblemas y procedimiento de cálculo de cotas inferiores.

A continuación se determinará el planteamiento y metodología elegida.

3.1. Planteamiento

3.1.1. Relajación del problema

Considérese el problema (P) con $C = [c_{ij}]$ la matriz de costes. La relajación del problema, consiste en la eliminación de la restricción (4):

$$\text{(AP)} \quad \text{Minimizar} \quad \sum_{i,j \in V} c_{ij} x_{ij} \tag{1}$$

sujeto a

$$\sum_{j \in V} x_{ij} = 1 \quad \forall i \in V \setminus \{j\} \tag{2}$$

$$\sum_{i \in V} x_{ij} = 1 \quad \forall j \in V \setminus \{i\} \tag{3}$$

$$x_{ij} = \{0, 1\} \quad \forall i, j \in V \tag{5}$$

de modo con T factible las variables tomarían los valores

$$x_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \in I \end{cases}$$

con E las aristas incluidas en la solución e I las que no lo están.
La elección de la eliminación de la restricción (4) se basa en la volumetría de restricciones a añadir.

A priori, podrán formarse ciclos de menor longitud en la solución. Sin embargo, se evitarán de manera *manual* como se verá más adelante.

Encontrar una solución óptima al problema relajado (AP) proporcionará una cota superior para $v(P)$,es decir,$v(AP) \geq v(P)$,por ser este menos restrictivo que el problema original.

Además, un modo de obtener una cota inferior de $v(P)$ en (AP) será mediante la reducción de la matriz de costes de la siguiente forma:

Definición 3.1.1. *Una fila (columna) de una matriz se dice* ***reducida*** *si sus elementos son no negativos y contiene al menos un cero.*

Definición 3.1.2. *Una matriz se dice* ***reducida*** *si cada fila y cada columna es reducida.*

Sea $C = [c_{ij}]$ la matriz de costes asociada a un grafo ponderado. Se reducen las filas de modo que para cada fila i, $u_i = min\{c_{ij} : j \in V\}$ luego $\hat{c}_{ij} := c_{ij} - u_i \ \forall i, j \in V$ luego, $\hat{C} := [\hat{c}_{ij}]$.

De igual manera para la reducción de columnas. Para cada columna j, $v_j = min\{\hat{c}_{ij} : i \in V\}$ por lo que $\bar{c}_{ij} := \hat{c}_{ij} - v_j$. Así, se obtiene la matriz reducida por filas y columnas $C_{red} := [\bar{c}_{ij}]$.
Veamos un ejemplo sobre una matriz C:

$$C = \begin{bmatrix} \infty & 8 & 25 & 3 \\ 8 & \infty & 12 & 4 \\ 25 & 12 & \infty & 15 \\ 3 & 4 & 15 & \infty \end{bmatrix} \overset{u=[3,4,12,3]}{\rightarrow} \hat{C} = \begin{bmatrix} \infty & 5 & 22 & 0 \\ 4 & \infty & 8 & 0 \\ 13 & 0 & \infty & 3 \\ 0 & 1 & 12 & \infty \end{bmatrix} \overset{v=[0,0,8,0]}{\rightarrow} C_{red} = \bar{C} = \begin{bmatrix} \infty & 5 & 14 & 0 \\ 4 & \infty & 0 & 0 \\ 13 & 0 & \infty & 3 \\ 0 & 1 & 4 & \infty \end{bmatrix}$$

El valor $LB = \sum_{i \in V} u_i + \sum_{j \in V} v_j$ será una cota inferior de $v(P)$:

Proposición. 3.1.3. *LB es una cota inferior del problema. O lo que es lo mismo,* $\sum_{i\in V} u_i + \sum_{j\in V} v_j \leq \sum_{i,j\in V} c_{ij}x_{ij}$ *con la definición de* u_i, v_j *anterior.*

Demostración. La obtención de una solución factible supone la elección de exactamente un elemento por fila y por columna de la matriz de costes 2.3.2. Sea u_i el mínimo de cada fila de C, v_j el mínimo de cada columna de $\hat{C}$ y $z(T)$ el coste total con $T = [(i_1, i_2), \ldots, (i_{k-1}, i_k)]$ una solución factible.
Si $k = n$ *entonces* $i_k i_1$:
$z(T) = \sum_{k=1}^{n} c_{i_k i_{k+1}} = \sum_{k=1}^{n}(c_{i_k i_{k+1}} - u_k + u_k) = \sum_{k=1}^{n} \hat{c}_{i_k i_{k+1}} + \sum_{k=1}^{n} u_k =$
$\sum_{k=1}^{n}(\hat{c}_{i_k i_{k+1}} - v_{k+1} + v_{k+1}) + \sum_{k=1}^{n} u_k = \sum_{k=1}^{n}(\bar{c}_{i_k i_{k+1}} + v_{k+1}) + \sum_{k=1}^{n} u_k = \sum_{k=1}^{n} \bar{c}_{i_k i_{k+1}} +$
$\sum_{k=1}^{n} u_k + \sum_{k=1}^{n} v_{k+1} = \sum_{k=1}^{n} \bar{c}_{i_k i_{k+1}} + \sum_{k=1}^{n} u_k + \sum_{k=1}^{n} v_k$

Como $\bar{c}_{i_k i_{k+1}} \geq 0 \Longrightarrow z(t) \geq \sum_{k=1}^{n} u_k + \sum_{k=1}^{n} v_k \forall\ T$ solución factible. □

3.1.2. Regla de Ramificación Little

En este apartado se explicará la técnica de ramificación escogida así como el orden del siguiente subproblema a ramificar y la construción de la cota inferior de cada uno de ellos.

A lo largo de la historia del TSP se desarrollaron diversas técnicas de ramificación considerando una *buena* regla la que genera pocos hijos de cualquiera de los nodos del árbol de decisión y por tanto que deja excluídas muchas soluciones no óptimas. La primera regla expuesta por John D. C. Little, Katta G. Murty, Sweeney y Karel genera únicamente dos hijos por cada nodo del árbol de decisión y sigue siendo la más efectiva hasta el momento.

Esta regla será por tanto la que se exponga a continuación:

Definición 3.1.4. *Sea el subproblema relajado* AP_k *(iteración k) y su matriz de costes reducida dada por* $C_{red} := [\bar{c}_{ij}]$, *se define* ***penalización*** *sobre cada* $\bar{c}_{ij} = 0$ *como:*

$$P_{ij} = min\{\bar{c}_{ih} : h \in V \setminus \{j\}\} + min\{\bar{c}_{hj} : h \in V \setminus \{i\}\} \tag{3.1}$$

Una arista $(r, s) \in A \setminus (E_k \cup I_k)$ será elegida si $P_{rs} = max\{P_{ij} : \bar{c}_{ij} = 0\}$ es decir, encontrar la arista (r, s) consistirá en buscar una arista que no fue escogida todavía y que tiene el cero con la penalización más alta.
Esta elección se realiza de modo que el algoritmo se centra en estudiar primero los subproblemas de menor coste.

Definición 3.1.5. *La* ***penalización fila*** *se define como* $P_i = min\{\bar{c}_{ih} : h \in V \setminus \{j\}\}$.

Definición 3.1.6. *La* ***penalización columna*** *como* $P_j = min\{\bar{c}_{hj} : h \in V \setminus \{i\}\}$.

En caso de haber varios ceros con la misma P_{ij}, se escogerá sin pérdida de generalidad el que tenga mayor penalización fila.

Una vez elegida la arista, se ramificará generando dos hijos del nodo k; los nodos $k+1$ y $(k+1)'$ definidos como:

$$E_{k+1} = E_k \cup \{(r, s)\} \quad I_{k+1} = I_k$$
$$E_{(k+1)'} = E_k \quad I_{(k+1)'} = I_k \cup \{(r, s)\}$$

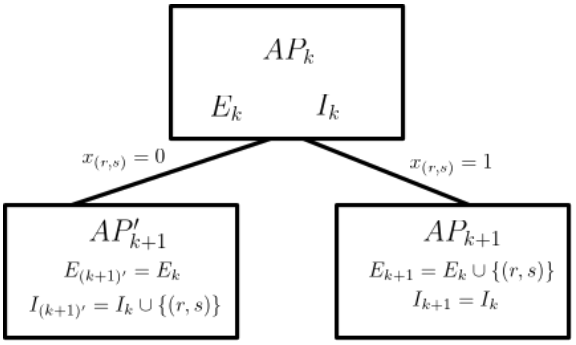

Figura 3.1: Ramificación de un nodo

Cada uno de estos nuevos nodos, tendrán asociados una nueva matriz de costes reducida y una cota inferior de la solución parcial. Dependiendo de si se esta en la rama de la derecha o en la rama de la izquierda, estos elementos se calcularán de manera diferente:

- Si $x_{ij} = 1 \Longrightarrow (i,j) \in E_k$
  - *Matriz de costes*

    1. *Eliminación de fila y columna* Se sustrae la fila i y la columna j para asegurarse de que no se elijan más aristas que salgan de i o lleguen a j.
    2. *Elemento* $c_{ji} = \infty$ Para evitar la elección de la misma arista en el otro sentido
    3. *Evitar ciclos de menor longitud* Si después de incluir la arista (i,j), incluir otra arista (k,l) formaría un ciclo de menor longitud en la solución, se impone $c_{kl} = \infty$. Por ejemplo si se tiene la cadena [3 2 5 7] se impondrá $c_{73} = \infty$.
    4. *Reducción de la matriz* Después de las transformaciones anteriores, se vuelve a reducir la matriz en caso de que sea necesario que tendrá un coste de reducción asociado $coste_{red}$.
  - *Cota inferior (LB)* Será la cota inferior del nodo anterior más las reducciones de fila y columna que resulten del último paso de la transformación en la matriz de costes, es decir , $LB_{k+1} = LB_k + coste_{red}$
- Si $x_{ij} = 0 \Longrightarrow (i,j) \in I'_{k+1}$
  - *Matriz de costes* Se impone $c_{ij} = \infty$ y se reduce la matriz cuyo coste de reducción coincidirá con P_{ij}.
  - *Cota inferior (LB)* Será la LB del nodo anterior más la penalización del 0 correspondiente por no haber escogido esa arista:
    $LB'_{k+1} = LB_k + P_{ij}$

.

Ahora, será necesario un criterio que determine que nodo de los dos nodos hijos generados será el siguiente en ramificar.

3.1.3. Exploración del árbol de decisión

Se sigue un algoritmo de búsqueda en profundidad (Definición 1.1.17).

Definamos primero la terminología empleada en los problemas de ramificación:

Definición 3.1.7. *Un **nodo vivo** es un nodo del espacio de soluciones del que aún no se han generado todos sus hijos.*

Definición 3.1.8. *Un **nodo muerto** es un nodo del que no se van a generar más hijos debido a que:*

- *No hay más hijos posibles*
- *No producirá una solución mejor que la actual*

Definición 3.1.9. *Un **nodo en expansión** es un nodo del que se están generando hijos.*

Inicialmente, se hará una búsqueda en profundidad "a ciegas" utilizando el método *LIFO*(*Last In, First Out*). En este método, se van expandiendo los nodos siguiendo la rama en la que $x_{ij} = 1$ $\forall i, j \in V$ hasta que esta rama muera porque todos los vértices fueron *visitados* ($|I_n| = \emptyset$ y $|E_n| = n$), obteniendo una primera solución factible T y su LB correspondiente, que será una cota superior de $v(P)$. A esta cota superior se le denominará Z .
Para explorar los nodos de las ramas en las que $x_{ij} = 0$, se aplicará de nuevo el proceso anterior para el nodo vivo que tenga menor LB asociada con la salvedad de que se parará de explorar cuando el nodo a ramificar muera por tener $LB \geq Z$. Por el contrario si en esta exploración se encuentra una solución factible de menor coste, este será la nueva Z.

El proceso de ramificación terminará cuando todos los nodos vivos hayan sido explorados o desechados. Obteniendo la solución óptima y su coste asociado.

Observación 3.1.10. *Cuando se parte de un nodo en el que ya se han añadido $n-2$ aristas, solo se puede cerrar ciclo con las dos aristas restantes por lo que se escogerán ambas al tiempo.*

La subdivisión del conjunto de posibles soluciones de la región factible del problema P, se representará como se ve en la figura siguiente:

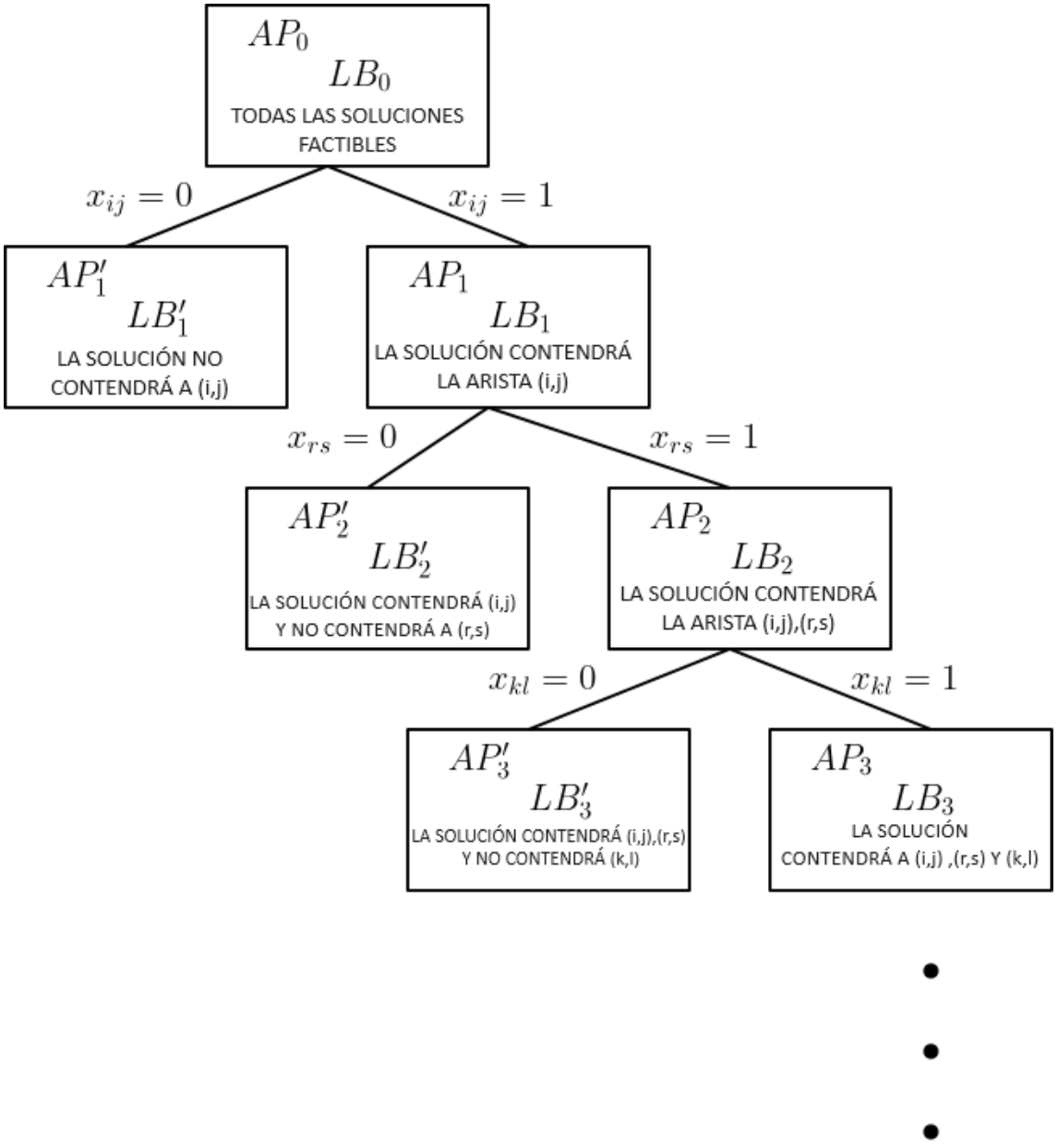

Figura 3.2: Ramificación en árbol del problema

En la figura anterior se expanden en este orden los problemas relajados $AP_0 \longrightarrow AP_1 \longrightarrow \cdots \longrightarrow AP_{n-2}$ y se crean los nodos vivos AP'_1, AP'_2, AP'_{n-2}.En las siguientes iteraciones el nodo de menor LB'_k pasa a ser el nuevo nodo en expansión.

Algorithm 2 Algoritmo Branch and Bound

INPUT: Matriz de costes C y problema P

NODO RAIZ (PASO 0):

1. Reducir la matriz C por filas y columnas.
2. Calcular una cota inferior LB_0 sumando las reducciones de filas y columnas.
3. Inicializar los conjuntos $E = \emptyset$ e $I = \emptyset$;

RESTO DE NODOS (PASO k):
1. Calcular la penalización P_{ij} para cada cero de la matriz de costes.
2. Escoger la arista con el cero de mayor penalización. Si hay más de uno, escoger la de mayor penalización fila.
3. Ramificar el árbol de búsqueda en el nodo calculando las cotas inferiores según corresponda:

1. **Si la arista** (i, j) **no está en la solución,** $x_{ij} = 0$**:**

   a) Incrementar el coste del nodo anterior en la penalización del cero correspondiente, $LB'_k = LB_{k-1} + P_{ij}$.

   b) Asignar un valor arbitrariamente grande (∞) en c_{ij} y reducir la matriz resultante.

   c) Añadir (i, j) a I'_k.

2. **Si la arista** (i, j) **está en la solución,** $x_{ij} = 1$**:**

   a) Eliminar la fila i y la columna j de la matriz de costes. En la práctica, poner ∞ en esas posiciones.

   b) Asignar un valor arbitrariamente grande (∞) en c_{ji} y asegurarse de que no se formen ciclos de menor longitud asignando valores ∞.

   c) Reducir la matriz.

   d) Calcular la cota inferior de dicha reducción: $LB_k = LB_{k-1} + \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j$.

   e) Añadir (i, j) a E_k.

4. Continuar la ramificación en la rama en la que $x_{ij} = 1$ repitiendo el proceso 1-4 hasta obtener una matriz 2×2.
5. Reducir la matriz si fuese necesario e incrementar la cota. Ambas aristas con coste cero formarán parte de la solución.
6. La última cota inferior obtenida será una cota superior Z del problema P.
7. Si $LB'_k \geq Z$ en todas las ramas, se acaba el proceso y el ciclo obtenido en el paso 6 será la solución óptima.
8. En otro caso, mientras $LB < Z$ repetir el proceso 1-6 para el nuevo nodo raíz. Si en el proceso se encuentra una nueva solución factible con menor coste, se actualizará al igual que el nuevo coste.
El algoritmo termina cuando todos los nodos mueran por tener $LB \geq Z$.
=0

3.2. Implementación en Matlab

Este método fue implementado en Matlab del modo que se muestra a continuación. Será expuesto el orden de llamada a las distintas funciones, una breve descripción de cada una de las funciones asi como de sus 'inputs' y 'outputs':

Esquema de llamadas:

- `function [mejorsol, costesol, tiempo_branch, tiempo] = BB_TSP(matrizcostes)`
  - `function [LB, E, Eord, LB_I_lista, matriz_reducida_lista, E_lista, Eord_lista] = BB_base(matrizred, LB, Z, E, Eord)`
    - `function [max_epenalizacion,fila,columna]=calcula_epenalizacion(matriz)`
    - `function [Eord, e1, e2] = verificar_exciclo(Eord, arista_nueva)`
    - `function [matrizred, costered] = reducematriz(matrizcostes)`
  - `function [matrizred, costered] = reducematriz(matrizcostes)`

Breve resumen de cada función:

- **BB_TSP**: Función principal que implementa el método de branch and bound, tan solo es necesaria una matriz de costes asociada a un grafo ponderado.
- **BB_base**: Función que a partir de un nodo raíz con su matriz reducida y cota inferior que da dicha solución hasta el momento, hace la ramificación a derecha hasta encontrar una solución completa (aunque no necesariamente óptima).

  En caso de tener más de dos input, ya existe una solución completa y se está ramificando a derecha nodos vivos previamente almacenados.
- **reducematriz**: Función que, a partir de una matriz de costes, obtiene la matriz reducida y el coste de dicha reducción (suma de los valores para la reducción de filas y columnas).
- **calcula_penalizacion**: Función que a partir de una matriz, calcula la máxima penalización además de las penalizaciones fila y columna.
- **verificar_exciclo**: Función que a partir de una serie de cadenas de aristas ya añadidas a la solución, comprueba como poder "pegar" al añadir una arista nueva para evitar que se cree un ciclo de menor longitud. Tiene como objetivo "colocar" esa nueva arista añadida en la cadena correspondiente (uniendo dos cadenas si se diese el caso) además de reconocer los extremos de los nuevos ciclos de longitud menor a n que se deben evitar.

Descripción de las variables empleadas:

- **matrizcostes** : Matriz simétrica de tamaño $n \times n$.
- **mejorsol** : Vector fila con las aristas que están incluidas en la solución
- **costesol** : Valor con el coste de la solución
- **tiempo_branch** : Valor en segundos del "branching a derecha"
- **tiempo** : Valor en segundos correspondiente al tiempo de ejecución
- **matrizred** : Matriz simétrica reducida de tamaño $n \times n$
- **costered** : Valor del coste de la reducción
- **LB** : Cota inferior de v(P)
- **Z** : Cota superior de v(P)
- **E** : Matriz $n \times 2$ que contiene las aristas incluidas en la solución

- **LB_I_lista**: Vector fila que almacena las cotas inferiores correspondientes a las soluciones parciales de los nodos de la izquierda
- **matriz_reducida_lista** : Array que tendrá almacenadas las matrices de los nodos de la izquierda
- **E_lista**: Array con las aristas que se van incluyendo para ir creando la solución parcial, por tanto, en el orden de inclusión
- **Eord_lista**: Array con las aristas de E ordenadas en cadenas disjuntas(en caso de no estar la solución completa)
- **Eord** : Array que contiene en las celdas una matriz $k \times 2$, donde están concatenadas k aristas
- **e1,e2**: Arista a evitar para que no forme ciclo
- **arista_nueva** : Vector 2×1 que se corresponde con la arista que es añadida a la solución en cada paso.
- **max_penalizacion**: Valor de la penalización máxima y, por tanto, la elegida
- **fila**: Índice de la fila de la matriz que contiene el 0 con mayor penalización
- **columna** : Índice de la columna de la matriz que contiene el 0 con mayor penalización

3.3. Ejemplo ilustrativo para sTSP con Branch and Bound

A continuación se muestra un grafo con 4 nodos y pesos en sus aristas para ilustrar el procedimiento de este método:

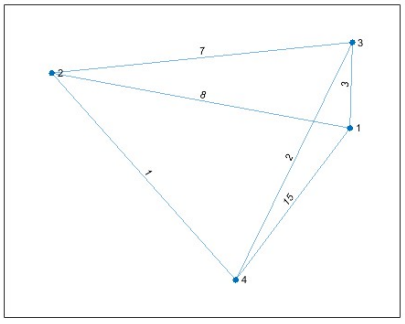

Figura 3.3: Grafo completo de ejemplo con 4 vértices

El grafo tiene asociada la matriz de costes siguiente:

$$C^0 = C = \begin{bmatrix} \infty & 8 & 3 & 15 \\ 8 & \infty & 7 & 1 \\ 3 & 7 & \infty & 2 \\ 15 & 1 & 2 & \infty \end{bmatrix}$$

Observación 3.3.1. *La notación empleada para las matrices será, C^k para las matrices asociadas a los nodos de la derecha y $C^{k\prime}$ para los de la izquierda. El superíndice k representa el nivel en el que se encuentra asumiendo que el nodo raíz está en el nivel 0. En caso de que la matriz sea reducida tendrá el subíndice red.*

El primer paso será obtener una matriz reducida:
Para ello, se calcula el mínimo de cada fila obteniéndose $u = (3, 1, 2, 1)$ y restando en cada fila:

$$\hat{C}^0 = \begin{bmatrix} \infty & 5 & 0 & 12 \\ 7 & \infty & 6 & 0 \\ 1 & 5 & \infty & 0 \\ 14 & 0 & 1 & \infty \end{bmatrix}$$

Se hará lo mismo con las columnas de $\hat{C}^0$, siendo $v = (1, 0, 0, 0)$ y restando en cada una de las columnas:

$$C^0_{red} = \begin{bmatrix} \infty & 5 & 0 & 12 \\ 6 & \infty & 6 & 0 \\ 0 & 5 & \infty & 0 \\ 13 & 0 & 1 & \infty \end{bmatrix}$$

$LB_0 = \sum u_i + \sum v_j = 3 + 1 + 2 + 1 + 1 + 0 + 0 + 0 = 8$.
Ahora se calculará la penalización de cada uno de los ceros. Esta será la suma del valor mínimo de la fila y la columna en la que se encuentre.

$$C^0_{red} = \begin{bmatrix} \infty & 5 & 0^{(5+1)} & 12 \\ 6 & \infty & 6 & 0^{(6+0)} \\ 0^{(0+6)} & 5 & \infty & 0^{(0+0)} \\ 13 & 0^{(1+5)} & 1 & \infty \end{bmatrix} = \begin{bmatrix} \infty & 5 & 0^{(6)} & 12 \\ 6 & \infty & 6 & 0^{(6)} \\ 0^{(6)} & 5 & \infty & 0^{(0)} \\ 13 & 0^{(6)} & 1 & \infty \end{bmatrix}$$

En la primera matriz se muestran los ceros con la penalización desglosada en penalización fila y penalización columna, ya que como varios 0 tienen la misma penalización (en la matriz de la derecha), se escogerá el 0 con mayor penalización fila.
En este caso, teniendo en cuenta todo lo anterior, la primera arista que ramificará el árbol será la (2,4) y se seguirá el orden que se muestra:

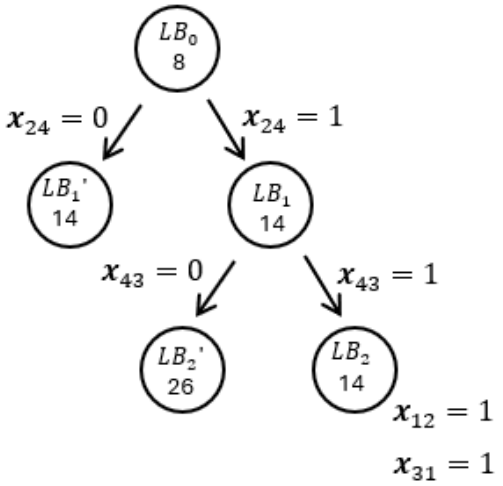

Figura 3.4: Árbol correspondiente a la Figura 3.3

En la Figura 3.4 se representan las cotas inferiores en cada paso como LB_k o LB'_k (el k representa la *profundidad* en la que se encuentra en el árbol) según corresponda a ramas de la derecha o de la izquierda respectivamente que son calculadas de distinta manera como se puede mostrar en el algoritmo que se muestra en el apartado 3.1.2.

Ilustremos de forma práctica el procedimiento:

$$C^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} \infty & 5 & 0 & 12 \\ \cancel{6} & \cancel{\infty} & \cancel{6} & \cancel{0} \\ 0 & 5 & \infty & 0 \\ 13 & 0 & 1 & \infty \end{bmatrix} \end{array} = \begin{array}{c} \\ 1 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \infty & 5 & 0 \\ 0 & 5 & \infty \\ 13 & \infty & 1 \end{bmatrix} \end{array}$$

Como la arista $(2,4)$ ya está en el recorrido, la arista $(4,2)$ no podrá hacerlo y por ello se tomará artificialmente un coste arbitrariamente grande ($c_{42} = \infty$).
Se comprueba que la matriz resultante sea reducida. La segunda columna y la tercera fila no tienen ningún 0 por lo que se extraerá 5 y 1 respectivamente. Quedando así la cota inferior
$LB_1 = 8 + 5 + 1 = 14$ y

$$C^1_{red} = \begin{array}{c} \\ 1 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \infty & 0 & 0 \\ 0 & 0 & \infty \\ 12 & \infty & 0 \end{bmatrix} \end{array}$$

La cota inferior LB'_1, casualmente toma el mismo valor que LB_1 pero no es obtenida de la misma forma. Esta cota, es la suma de LB_0 y 6 (penalización de no escoger el 0 de la posición $(2,4)$ de la matriz) y tiene la matriz asociada siguiente:

$$C^{1\prime} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} \infty & 5 & 0 & 12 \\ 6 & \infty & 6 & \infty \\ 0 & 5 & \infty & 0 \\ 13 & 0 & 1 & \infty \end{bmatrix} \end{array}$$

Se sigue ramificando por el nodo de la derecha hasta obtener una primera solución sea o no la óptima. Se calculan las penalizaciones de los ceros de C^1_{red}, se escogerá la siguiente arista donde el 0 tenga la mayor penalización y se repetirá el proceso anterior.

$$C^1_{red} = \begin{bmatrix} \infty & 0^{(0+0)} & 0^{(0+0)} \\ 0^{(0+12)} & 0^{(0+0)} & \infty \\ 12 & \infty & 0^{(12+0)} \end{bmatrix} \Longrightarrow C^2 = \begin{array}{c} \\ 1 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \infty & 0 & 0 \\ 0 & 0 & \infty \\ \cancel{12} & \cancel{\infty} & \cancel{0} \end{bmatrix} \end{array} = \begin{array}{c} \\ 1 \\ 3 \end{array} \begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{bmatrix} \infty & 0 \\ 0 & \infty \end{bmatrix} \end{array}$$

En este caso, el coste de la arista $(3,4)$ no aparece, ya que se sustrajo la columna 4 por lo que para que en la solución parcial $2-4-3$ se cumpla la condición (4) de P , será necesario que $(3,2)$ no forme parte del ciclo y, por tanto, que tenga un coste arbitrariamente grande.

La matriz resultante es de tamaño, 2×2 luego ambas aristas con coste distinto de infinito formarán parte de la solución, las aristas $(3,1)$ y $(1,2)$. La cota superior será $Z = 14$.
Se cumple que $LB'_k \geq Z$ con $k \in 1,2$ luego se habrá obtenido la solución óptima [1 2 4 3 1] cuyo coste será 14.

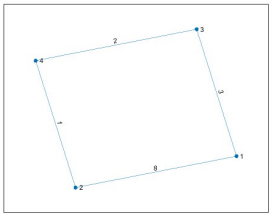

Figura 3.5: Solución óptima del ejemplo

Capítulo 4

Lin-Kernighan

Teniendo en cuenta la clasificación explicada en la sección 2.5 sobre los métodos heurísticos, el más destacable de búsqueda local es el algoritmo $k-opt$ que será explicado en más detalle en lo que sigue. Al tratarse de un algoritmo de mejora local, no puede demostrar la optimalidad de la solución, pero si proporcionar una confianza estadística adecuada para las aplicaciones prácticas ya que, cuenta con tiempos de ejecución relativamente bajos. Esta técnica está planteada para aplicarse sobre instancias del TSP simétrico.[9, 11, 14]

Una técnica heurística de búsqueda local en un problema de Optimización Combinatoria sigue las siguientes instrucciones:

1. Generar una solución factible al azar, es decir, un ciclo hamiltoniano T que satisface las restricciones del problema P (Problema 2.3).
2. Haciendo transformaciones sobre T trata de buscar una solución factible T' que mejore el coste de T.
3. Si se encuentra sustitución de T cumpliendo $f(T') < f(T)$ donde f es la función objetivo a minimizar luego, $T = T'$ y se repite el paso anterior.
4. Si no se encuentra ninguna mejora posible, T es un óptimo local y se repite el proceso desde el principio.

La primera heurística utilizada para la realización del paso 2, fue el intercambio $2-opt$ propuesto por *Croes* en 1958 que consiste en el intercambio de dos aristas por otra dos que mejoren la solución factible inicial. Años más tarde, se definió una generalización de la misma: el intercambio $k-opt$.

Definición 4.0.1. *Un* k***-intercambio*** *sobre una solución factible* T *es un intercambio de* k *aristas de* T *por otras* k *que no pertenecen a* T *y tal que, se obtenga una nueva solución* T' *factible.*
Se dice que T *Y* T' *son* ***ciclos vecinos****.*
Se dice $k-$ ***óptimo(o*** $k-opt$***)*** *si* T' *tiene menor coste que* T*.*

Cuanto mayor sea el valor de k, mayor probabilidad habrá de que el ciclo final T' sea óptimo. Sin embargo, el esfuerzo computacional aumenta rápidamente al aumentar, k por lo que es difícil saber de antemano qué k es el mejor para satisfacer que la solución tenga una *buena calidad* y a la vez que tenga un *buen tiempo* de ejecución. En la práctica, se suele trabajar con $2-opt$ y $3-opt$ ya que está demostrado que para valores de k superiores el tiempo extra de computación no compensa en relación con la mejora de los resultados. Este último enunciado se cita en alguna de las referencias empleadas, sin embargo, aunque parece lógico y en la implementación se evaluará dicha ´´obviedad" realizando $2-opt$ iterativos, no hay documentos al alcance de todos con pruebas que así lo determinen.

El algoritmo de Lin-Kernighan crea una nueva versión del algoritmo $k-opt$ eliminando el inconveniente de tener que imponer el k de antemano y, a pesar de ser planteado en el año 1973, se sigue considerando la mejor técnica heurística disponible para la resolución del TSP simétrico.[9, 11, 14]

Además, el algoritmo original tuvo diversas modificaciones para tratar de mejorar los resultados y eficiencia del método. Alguna de ellas se pueden encontrar en los documentos que se referencian más abajo ,pero este trabajo está únicamente centrado en el algoritmo base original (LK) así como algunos refinamientos que se añadieron posteriormente sobre este.

4.1. Planteamiento

Esta técnica va decidiendo cuál será el valor de k en cada iteración, intentando escoger el mayor k posible para dar como resultado el ciclo de menor peso. Es decir, se inicializa con una solución factible y se comprueba la existencia de ciclos vecinos de menor costo considerando un conjunto creciente de intercambios potenciales, comenzando con $k = 2$ o $k = 3$.

Sea A el conjunto de todas las aristas y T el recorrido inicial. El algoritmo trata de encontrar dos conjuntos de aristas, $X = \{x_1, x_2, \ldots, x_k\}$ e $Y = \{y_1, y_2, \ldots, y_k^*\}$ tales que, si las aristas de X se eliminan de T y se sustituyen por las aristas de Y pertenecientes a $A - T$, el recorrido T' tiene menor peso que T. Los conjuntos X e Y comienzan vacíos y se construyen elemento a elemento, de modo que en el paso i se añaden x_i e y_i a los conjuntos respectivamente. Las aristas de X se denominan **aristas de salida** y las de Y **aristas de entrada**.

Observación 4.1.1. *No cualquier par de conjunto X,Y genera un k-intercambio. Esto se debe a las diversas formas de reconexión de las partes disjuntas que crea la eliminación de las aristas de X. Tomemos como ejemplo $x_1 = (i, j)$ y $x_2 = (k, l)$ aristas no adyacentes que forman parte de T. Si se eliminan del recorrido inicial, quedarán dos partes disjuntas (A y B) que deberán ser reconectadas. A continuación se muestran las posibles reconexiones:*

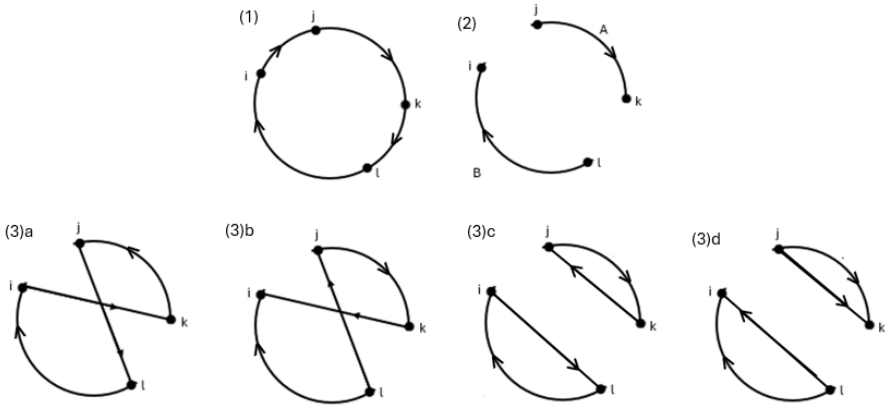

Figura 4.1: Movimiento 2-opt

En la Figura anterior se muestran cuatro maneras de reconectar (elegir y_1, y_2) las aristas eliminadas, pero, solo las opciones (3)*a y* (3)*b serán válidas (cambian el sentido de recorrido) para un intercambio* $2 - opt$ *ya que la* (3)*c y* (3)*d violará la restricción (4) de no formación de ciclos disjuntos.*

A medida que aumentan las aristas a intercambiar, aumentan las posibilidades de reconexión y, por tanto, la complejidad del algoritmo. Un ejemplo de posible intercambio $3 - opt$ *se vería del siguiente modo:*

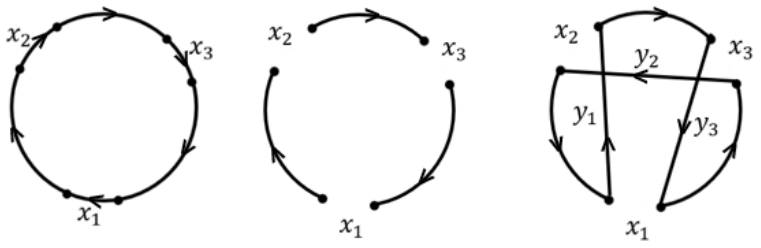

Figura 4.2: Movimiento 3-opt donde las aristas x_1, x_2, x_3 son reemplazadas por las y_1, y_2, y_3

Los criterios que se siguen para introducir aristas en los conjuntos X e Y son los siguientes:

- ***Criterio de intercambio secuencial***: x_i e y_i tienen que compartir un punto final, al igual que y_i y x_{i+1}. Si t_1 denota uno de los extremos de x_1, se tiene en general $x_i = (t_{2i-1}, t_{2i})$, $y_i = (t_{2i}, t_{2i+1})$ y $x_{i+1} = (t_{2i+1}, t_{2i+2})$ $\forall i \geq 1$.
  La secuencia $(x_1, y_1, x_2, y_2 \dots x_k, y_k)$ corresponde con un camino que alterna aristas de X e Y denominado **ciclo alterno** (Figura 4.3) que será representado por la variable XY en lo que sigue.

  No siempre se cumple que un intercambio de aristas sea secuencial, como es el caso en la Figura 4.4 donde no se pueden numerar las aristas para que el intercambio sea secuencial.

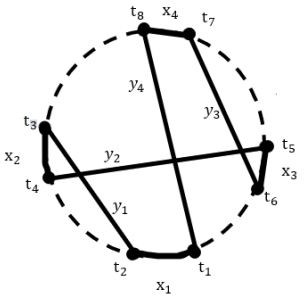

Figura 4.3: Ciclo alterno con 8 vértices

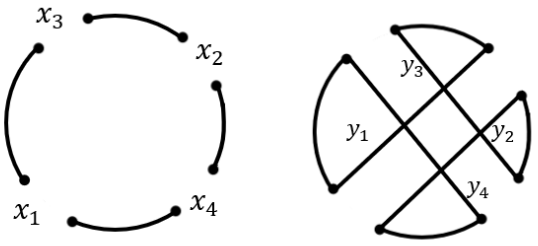

Figura 4.4: Intercambio no secuencial con $k = 4$

- ***Criterio de Factibilidad***: Se requiere que $x_i = (t_{2i-1}, t_{2i})$ $\forall i \geq 3$ se elija de modo que, si t_{2i} se une a t_1, el camino resultante sea un ciclo. Es decir, que se pueda detener el proceso generando $y_i^* = (t_{2i}, t_1)$ y obteniendo así una solución factible al intercambiar $X = \{x_1, x_2, \dots, x_i\}$ por $Y = \{y_1, y_2, \dots, y_i^*\}$.
  Una condición necesaria, pero no suficiente para que el intercambio de lugar a un recorrido, es que la secuencia creada sea cerrada; es decir, que exista $y_k^* = (t_{2k}, t_1)$.

- ***Criterio de Ganancia Positiva***: Cada intercambio de aristas proporcionará una ganancia $g_i = |x_i| - |y_i|$ donde $|x_i|$ y $|y_i|$ representan el coste de las aristas correspondientes.
  La elección de y_i se realiza de modo que la ganancia parcial $G_i = g_1 + g_2 + \dots + g_i$ sea positiva. De esta forma, cuando se encuentra un g_i negativo, por la aditividad de la ganancia, no será necesario la detención del proceso (Proposición 4.1.2).

- ***Criterio de disyuntividad***: Se requiere que los conjuntos X, Y sean disjuntos. Así, x_i no puede ser una arista que se unió en un paso anterior e y_i no puede ser una arista que fue

eliminada previamente.

- ***Criterio de parada***: El criterio de parada decide que no se añadirán parejas en os conjuntos X,Y cuando $G_i \leq G^*$.

Si el algoritmo termina con $G^* > 0$ se habrá encontrado un $k-opt$ que dará una solución factible T' con coste $f(T') = f(T) - G^*$. Luego, se podrá tomar $T = T'$ para poder seguir mejorando la solución.

En caso de terminar el algoritmo con $G^* = 0$ se aplicará una técnica de backtracking para garantizar el cumplimiento del criterio de ganancia positiva y/o el criterio de Factibilidad en caso de no encontrar un x_i o y_i que ofrezcan una ganancia positiva y/o un y_i^* que cierre el ciclo, se empleará una técnica de backtracking para estudiar si una elección (entre las válidas) distinta del último vértice escogido, solventa el problema. Es importante destacar que esta técnica, aunque puede resultar muy útil, incrementa el tiempo de ejecución y reduce la eficiencia, por lo que su uso estará limitado a solo si no encuentra un $k-opt$ a realizar.Esta técnica será detallada más adelante.

Proposición. 4.1.2. *Si una secuencia de números tiene una suma positiva, existe una permutación cíclica de estos números tal que cada suma parcial es positiva.*

Demostración. Sea la secuencia original $g = (g_1, g_2, \ldots, g_n)$ y sea $G = \sum_{i=1}^{n} g_i > 0$. Considerándose las sumas parciales como $G_i = \sum_{j=1}^{i} g_j \ \forall i = 1, 2, \ldots, n$ se define s como el índice más grande tal que G_{s-1} es mínimo. Sea la permutación cíclica de la secuencia comenzando desde el índice s: $(g_s, g_{s+1}, \ldots, g_n, g_1, g_2, \ldots, g_{s-1}) = (h_1, h_2, \ldots, h_n) = h$ para reducir la notación. Se pretende demostrar que $H_i = \sum_{j=1}^{i} h_j > 0 \ \forall i = 1, \ldots, n$

Representado, por tanto, $h_i = \begin{cases} g_{i+s-1} - s_i & \text{si } i = 1, \ldots, n-s+1 \\ g_{i-(n-s+1)} - s_i & \text{si } i = n-s+2, \ldots, n \end{cases}$

Siendo H_i cada una de las sumas parciales de la nueva secuencia de modo que:

$$H_1 = h_1 = g_s$$

$$H_2 = h_1 + h_2 = g_s + g_{s+1}$$
$$\vdots$$
$$H_{n-s+1} = h_1 + \cdots + h_{n-s+1} = g_s + g_{s+1} + \cdots + g_n$$

$$H_{n-s+2} = h_1 + \cdots + h_{n-s+2} = g_s + g_{s+1} + \cdots + g_n + g_1$$
$$\vdots$$
$$H_n = h_1 + \cdots + h_n = g_k + g_{s+1} + \cdots + g_n + g_1 + g_2 + \cdots + g_{s-1} = G$$

Se pretende demostrar que cada $H_i > 0 \ \forall i = 1, 2, \ldots n$

- $1 \leq i \leq n - s + 1$
  $H_i = \sum_{j=1}^{i} h_j = \sum_{j=s}^{s+i-1} g_j = G_{s+i-1} - G_{s-1} > 0$.
  G_{s-1} es el mínimo y $i + s - 1 \geq s$ luego se cumplirá la desigualdad por la elección de s.

- $n - s + 2 \leq i \leq n$
  $H_i = \sum_{j=1}^{i} h_j = \sum_{j=s}^{n} g_j + \sum_{j=1}^{i-(n-s+1)} g_j = (G_n - G_{s-1}) + G_{i-(n-s+1)}$
  $= G_n + (G_{i-(n-s+1)} - G_{s-1}) > 0$.
  G_n es la suma total que por definición es positiva. Además, se cumple que $(G_{i-(n-s+1)} - G_{s-1}) \geq 0 \ \forall i \in \{n - s + 2, \ldots, n\}$ por el mismo argumento que en el otro caso.

□

4.2. Comentarios sobre el algoritmo

Además de todos los criterios mencionados para mejorar la eficiencia, se añaden los siguientes refinamientos sobre el backtracking:

- El backtracking solo se realiza en el nivel $i = 2$ e inferior.
- Cada t_{2i+1} (el que determina la arista y_i) está escogido con el menor peso posible.
- Para reducir la búsqueda, para los vértices t_3 y t_5 en caso de ser necesaria la aplicación de backtracking, este se realizará sobre los 5 vecinos más cercanos, es decir, se estudiarán los 5 y_1 e y_2 de menor peso respectivamente.
- Se permite incumplir el criterio de intercambio secuencial en el nivel $i = 2$. Se permite que no exista una arista $y_2^* = (t_4, t_1)$, con la condición de que exista la arista $y_3^* = (t_6, t_1)$. De este modo, el proceso empezaría con un 3-intercambio en vez de con un 2-intercambio.

Este método, a pesar de ser como dije el mejor hasta el momento, tiene distintas interpretaciones según los distintos artículos que empleé para este trabajo, luego fue complicado escoger qué formaba parte del algoritmo original y qué no. El algoritmo básico inicial [15] deja demasiadas decisiones sin determinar, por lo que se hace bastante difícil su interpretación total. Por esta controversia, he interpretado y tomado las siguientes decisiones sobre los frentes que dejan abiertos:

- La referencia citada anteriormente, expone que deben eliminarse las aristas que están 'más fuera de lugar' por lo que se escogerá el y_i que tenga menor peso y, por tanto, que proporcione una mayor ganancia ($g_i = |x_i| - |y_i|$).
- Las aristas $x_i = (t_{2i-1}, t_{2i})$ se crean de manera que t_{2i} sea adyacente en T a t_{2i-1} por lo que sin tener en cuenta ninguna restricción, tiene dos posibles opciones (a veces pudiendo estar unívocamente determinada por el Criterio de Factibilidad anteriormente descrito). Para las que ambas opciones son válidas, en la referencia [10] se explica que para x_1, x_2, x_3, el vértice t_{2i} se determinará aleatoriamente (en caso de mala elección se realizará backtracking sobre el vértice no utilizado) pero a partir de x_4 se escogerá la de peso máximo y, por tanto, de nuevo que proporcione mayor ganancia.
- Las aristas y_i $\forall i \geq 2$, se crearán con la certeza de que son capaces de "romper" un x_{i+1} válido, es decir, debe garantizarse la existencia de un y_{i+1}^* que cierre ciclo.
- Siempre (a excepción en el backtracking del paso $i = 2$) debe cumplirse el criterio de Factibilidad para encontrar intercambios secuenciales. Debe existir un y_i^* que cierre el ciclo en caso de no poder incrementar más el valor de k. Sin embargo, ese cierre, en caso de no tener ganancia positiva, no tiene que suponer una parada del algoritmo, simplemente no será una solución válida.
- En las dos referencias citadas, también aparece el enunciado "El algoritmo realiza un intercambio $2-opt$ o un $3-opt$ seguido de una secuencia (vacía o no) de $2-opt's$" que de primera mano sin justificación aparente puede resultar falsa.
  El algoritmo obliga a tener la opción de cierre (con ganancia o no) en cada paso i con el ciclo alterno $x_1y_1x_2y_2 \dots x_iy_i^*$, empezando desde $i = 2$ o $i = 3$.

  Supongamos que el algoritmo termina con el cierre en el paso $i + 1$:

  ciclo alterno $x_1y_1x_2y_2 \dots x_iy_ix_{i+1}y_{i+1}^*$
  aristas eliminar $X = \{x_1, \dots x_i, x_{i+1}\}$
  aristas añadir $Y = \{y_1, \dots, y_i, y_{i+1}^*\}$

  Si se hubiera hecho el intercambio de aristas en el paso i anterior, el intercambio habría sido:

ciclo alterno $x_1 y_1 x_2 y_2 \dots x_i y_i^*$
aristas eliminar $X = \{x_1, \dots x_{i-1}, x_i\}$
aristas añadir $Y = \{y_1, \dots, y_{i-1}, y_i^*\}$

Para llegar a la misma solución que habiendo hecho el intercambio realizado con el paso, $i+1$ basta intercambiar $\{y_i^*, x_{i+1}\}$ por $\{y_i, y_{i+1}^*\}$

$$\{x_1, \dots, x_{i-1}, x_i, x_{i+1}\} \longrightarrow \{y_1, \dots, y_{i-1}, y_i^*, x_{i+1}\} \longrightarrow \{y_1, \dots y_{i-1}, y_i, y_{i+1}^*\}$$

Sin embargo, como esta planeando el algoritmo LK, no hay garantía de que cada uno de estos 2-intercambios (o el posible primer 3-intercambio) sean con ganancia.

Ahora, se describe el algoritmo de Lin-Kernighan abordando todos los criterios y decisiones anteriormente descritas:

Algorithm 3 Algoritmo Lin -Kernighan(LK)

INPUT: Matriz costes C , T solución factible construído mediante una heurística constructiva como puede ser el *Vecino más próximo* 2.5.1

1. $i = 1 : G^* = 0$

   a) Escoger el nodo $t_1 \in V$ de manera aleatoria.

   b) $x_1 = (t_1, t_2)$ es una de las dos arista de modo que se cumple que t_2 sea adyacente en T a t_1.

   c) Ahora se debe escoger un nodo t_3 (no escogido anteriormente), de modo que $y_1 = (t_2, t_3) \notin T$ tenga peso mínimo y cumpliendo la condición $G_1 = g_1 = |x_1| - |y_1| > 0$. Si no existe tal t_3, ir al paso 3 (d).

2. Sea $i = i + 1$. Elegir $x_i = (t_{2i-1}, t_{2i})$ e y_i de la siguiente forma:

   (a) x_i se escoge de manera que si t_{2i} se une con t_1 con $(t_{2i}, t_1) \notin T$ este camino alterno genera un k-intercambio (se sabe que existe por el *Criterio de factibilidad* al haberlo construido en el paso anterior).
   Se compueba si este intercambio da un valor de ganancia positiva mejor que el que se tiene: sea y_i^* la arista que conecta estos nodos y sea $g_i^* = |x_i| - |y_i^*|$. Si $G_{i-1} + g_i^* > G^* \Longrightarrow G^* = G_{i-1} + g_i^*$ y $k = i$ produciendo, por tanto, un intercambio $k - opt$ que será utilizado en caso de no poder continuar con el proceso.

   (b) Se escoge y_i una arista con extremo t_{2i} cumpliendo:

      (a) *Criterio de disyuntividad* (no se había escogido antes este vértice).

      (b) *Criterio de ganancia positiva* ($G_i > 0$).

      (c) El y_i elegido sea capaz de romper una arista x_{i+1} ($\exists y_{i+1}^*$).

   Se termina este paso si no existe y_i o si $G_i \leq G^*$ (*Criterio de parada*).

3. Si $G^* = 0$ se aplica un backtracking limitado:

   (a) Repetir el paso (2)(b) para $i = 2$ y buscar otro y_2 tratando de incrementar su tamaño y satisfaciendo $G_2 = g_1 + g_2 > 0$.

   (b) Si en ninguna opción del paso anterior se obtiene beneficio, volver al paso (2)(a) para $i = 2$ y elegir otra posibilidad para x_2.

   (c) Si el paso anterior también falla ir al paso (1)(c) para probar distintos y_1.

   (d) Si el paso (c) no da mejora, se considera otra arista x_1 en el paso (1)(b).

   (e) Si (d) falla de nuevo, se selecciona otro t_1 en el paso (1)(a).

   El backtracking en el paso (3)(b) viola de forma temporal el *Criterio de Factibilidad* para aumentar la efectividad del método.

   El procedimiento termina cuando se probaron todos los n valores de t_1 posibles y no se encontró ninguna mejora.

4.3. Implementación en Matlab

La implementación en *Matlab* de este, fue lo que supuso mayor esfuerzo y gran parte del tiempo de la realización de este trabajo. La mayoría de implementaciones existentes son de modificaciones o no cumpliendo íntegramente el propósito que expuso Kernighan de aumentar el k lo máximo posible. Por ejemplo, en la referencia [10] en la implementación sustituye T por T' en cuanto se encuentra ganancia sin tratar de aumentar el número de aristas a intercambiar. Aun así, por falta de tiempo, quedaron puntualizaciones sin implementar.

Esquema de llamadas del programa:

`function [T_final,coste_inicial ,coste_final] = LK_TSP(matrizcostes)` La función anterior llamará de manera iterativa a lo siguiente:

- `function [T_nuevo,G_est,k,Xopt,Yopt] = LK_base(T, matrizcostes)`
  - `function E = soluc_aristas(T)`
  - `function [bool,sol] = es_tour(E,X_temp,Y_temp)`
  - `function coste = calcula_coste(T, matrizcostes)`
  - `function [T_nuevo,X,XY,back_4,bool_prov] = encontrar_t4(X,E,Y,XY,M_X,M_Y,back_4)`
    - `function [bool,sol] = es_tour(E,X_temp,Y_temp)`
      - `function [Eord,e1,e2] = verificar_exciclo(Eord,arista_nueva)`
  - `function [txi,sol_ahead,Y,XY,back_5,contador_5]= encontrar_t5(X,E,Y,XY,M_X,M_Y,cont,back_5)`
    - `function [bool,sol] = es_tour(E,X_temp,Y_temp)`
      - `function [Eord,e1,e2] = verificar_exciclo(Eord,arista_nueva)`
  - `function [Xnew,Ynew,XYnew,T_nuevo,txi] = aumentar_camino(T,M_X,M_Y,X,Y,XY,txi,sol_ahead)`
    - `function [bool,sol] = es_tour(E,X_temp,Y_temp)`
      - `function [Eord,e1,e2] = verificar_exciclo(Eord,arista_nueva)`

Breve resumen de cada función:

- **LK_TSP**: Función que a partir de una matriz de costes de un grafo ponderado genera una solución factible por un método constructivo para luego, tratar de mejorarla iterativamente con el método de Lin-Kernighan.
- **LK_base**: Función principal que implementa el método de Lin-Kernighan con bracktracking, es necesaria una solución inicial factible T y una matriz de costes asociada a un grafo.
- **soluc_aristas**: Función que a partir de la solución factible T $n \times 1$ (donde n es el nº de vértices/ciudades) que contiene los vértices ordenados del recorrido extrae una matriz $n \times 2$ con las aristas que forman parte de dicho recorrido
- **verificar_exciclo**: Función que a partir de una serie de cadenas de aristas ya añadidas a la solución, comprueba como poder "pegar" al añadir una arista nueva. Tiene como objetivo "colocar" esa nueva arista añadida en la cadena correspondiente.
- **es_tour**: Función que comprueba la existencia de un ciclo cerrado tras la eliminación de k aristas e introducción de otras k distintas, utilizando la función de pegado anteriormente descrita.
- **encontrar_t4**: Función que determina el vértice t_4 de x_2 de modo que pueda cerrar ciclo en caso de ser necesario y formar, por tanto y_2^*.
- **encontrar_t5**: Función que determina el vértice t_5 de y_2 de modo que rompa un x_3 verificando la existencia de y_3^*.

- **aumentar_camino**: Función que intenta aumentar el camino alterno $x_1y_1 \dots x_iy_i$ en una nueva pareja. Devuelve los conjuntos X,Y de aristas si es que ese nuevo intercambio genera ganancia.
- **calcula_coste**: Función que a partir de una solución factible T y la matriz de costes asociada al grafo ponderado calcula el coste del recorrido T.

Descripción de las variables empleadas:

- **matrizcostes** : Matriz simétrica de tamaño $n \times n$.
- **T_final** : Matriz $n \times 2$ que contiene una solución válida
- **coste_inicial**: Valor del coste de T
- **coste_final**: Valor que se corresponde con el coste de T_nuevo
- **T**: Vector fila que es solución factible del problema, contiene los vértices en el orden en que se visitan
- **T_nuevo**: Vector fila con la solución T mejorada tras haber intercambiado k aristas
- **G_est**: Valor de la mejora ganancia encontrada
- **Xopt** : Matriz $k \times 2$ que contiene las aristas a eliminar de T
- **Yopt** : Matriz $k \times 2$ que contiene las aristas a añadir de T
- **X** : Matriz $k \times 2$ que contiene en las filas las aristas a eliminar
- **Y** : Matriz $k \times 2$ que contiene en las filas las aristas a añadir
- **XY** : Vector que representa el ciclo alterno de longitud 2k
- **E** : Matriz $n \times 2$ que contiene las aristas existentes en T
- **Eord** : Array que contiene en las celdas una matriz $k \times 2$, donde están concatenadas k aristas
- **e1,e2**: Arista a evitar para que no forme ciclo esto en LK
- **M_X** : Matriz $n \times n$ que contiene los costes de las aristas que forman parte de la solución T. El resto de elementos toman valor 0
- **M_Y** : Matriz $n \times n$ que contiene los costes de las aristas que NO forman parte de la solución T. El resto de elementos toman valor INF
- **contador_i** : Valor que cuenta el número de t_{2i-1} que fueron probados
- **back_i** : Vector $1 \times j$ que contiene las posibles opciones válidas para t_i
- **sol_ahead** : Matriz $n \times 2$ que contiene una solución válida $x_1y_1x_2y_2x_iy_i^*$
- **bool** : Variable booleana que identifica si se encontró variable válida o no
- **txi** : Vértice que determina X. Calculado para que y_i rompa un x_{i+1}
- **yi_est_ahead**: Arista candidata a cerrar el ciclo
- **?_ahead** : Variables creadas para la comprobación de que y_i rompa un x_{i+1}.

A continuación se representa un ejemplo para tratar de ilustrar mejor este procedimiento.

4.4. Ejemplo ilustrativo para sTSP con Lin-Kernighan

Se escogió un grafo completo de 6 vértices representado por:

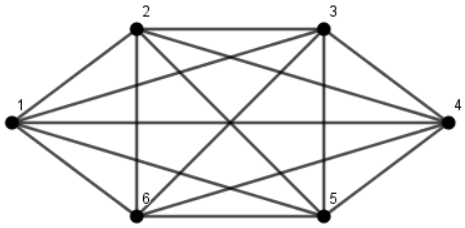

Figura 4.5: Grafo completo con 6 vértices

Por simplificar visualmente el grafo, no se representa el peso de cada una de las aristas, pero la matriz de costes asociada es la siguiente:

$$C = \begin{bmatrix} \infty & 1 & 8 & 6 & 4 & 3 \\ 1 & \infty & 2 & 9 & 3 & 7 \\ 8 & 2 & \infty & 1 & 12 & 5 \\ 6 & 9 & 1 & \infty & 4 & 13 \\ 4 & 3 & 12 & 4 & \infty & 1 \\ 3 & 7 & 5 & 13 & 1 & \infty \end{bmatrix}$$

Se escoge de manera aleatoria una solución factible inicial:

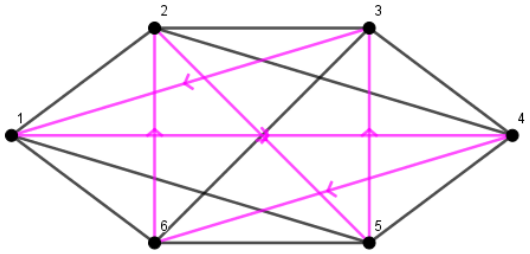

Figura 4.6: Posible solución del problema (en rosa)

Como se puede ver en la Figura, la solución inicial es el ciclo T=[1 4 6 2 5 3] y el valor de $f(T) = 6 + 13 + 7 + 3 + 12 + 8 = 49$.

Ahora se comienza el proceso iterativo con X = [ ] ,Y=[ ] y XY = [ ] :

1. **NIVEL 1**
   $G^* = 0$.
   Se escoge $t_1 = 1$ y $t_2 = 4 \Longrightarrow x_1 = (1, 4)$.

Ahora se elige $t_3 = 3$, nodo no escogido, no adyacente a t_2 en T y de peso mínimo. Luego $y_1 = (4, 3)$.
$g_1 = |x_1| - |y_1| = 6 - 1 = 5 > 0$
$G_1 = g_1 = 5$
$X = [(1, 4)]$, $Y = [(4, 3)]$ y XY=[1 4 3]

2. **NIVEL 2**
   Se debe probar si hay un x_2 de modo que t_4 no esté en XY, sea adyacente a t_3 y que $(t_4, t_1) \notin T$ pueda formar un ciclo.
   Se escoge $t_4 = 5 \Longrightarrow x_2 = (3, 5)$.
   Ahora, se debe calcular el coste de cerrar el ciclo con (t_4, t_1) en caso de que esto fuese necesario.
   Sea $y_2^* = (t_4, t_1) = (5, 1)$ por lo que $g_2^* = 12 - 4 = 8$ .Se comprueba que $G_1 + g_2^* = 5 + 8 = 13 > 0 = G^*$ y al cumplirse, se actualiza $G^* = G_1 + g_2^* = 13$ y $k = 2$ pudiéndose realizar, por tanto, un intercambio $2 - opt$.
   Se escoge el vértice $t_5 = 6$,nodo no escogido, no adyacente a t_4 en T y de peso mínimo: se define $y_2 = (5, 6)$. Este t_5 se escoge sabiendo que existe el candidato a $t_6 = 2$ que permite cerrar ciclo con (t_6, t_1) en el nivel 3.
   $g_2 = 12 - 1 = 11$
   $G_2 = g_1 + g_2 = 5 + 11 = 16 > 13 = G^*$ por lo que continuo el proceso.
   $X = [(1, 4), (3, 5)]$, $Y = [(4, 3), (5, 6)]$ y XY=[1 4 3 5 6]

3. **NIVEL 3**
   Se define $t_6 = 2$(determinado al escoger t_5) $\Longrightarrow x_3 = (6, 2)$. Continuando el proceso como en el paso anterior, $y_3^* = (2, 1)$ y $g_3^* = 7 - 1 = 6$ luego $G_2 + g_3^* = 16 + 6 = 22 > 13 = G^* \Longrightarrow G^* = G_2 + g_3^* = 22$ y $k = 3$. El 3-intercambio es un $3 - opt$.
   Como tiene 6 vértices no se puede seguir y, por tanto, la elección debe ser y_3^* cerrando el ciclo y acabando, por tanto, con un intercambio $3 - opt$.

Se cambia $T = T'$ con $f(T') = f(T) - G^* = 49 - 22 = 27$
$X = [(1, 4), (3, 5), (6, 2)]$, $Y = [(4, 3), (5, 6), (2, 1)]$ y XY = [1 4 3 5 6 2]
Se obtiene el grafo de la Figura siguiente en donde las aristas representadas en amarillo son las aristas añadidas:

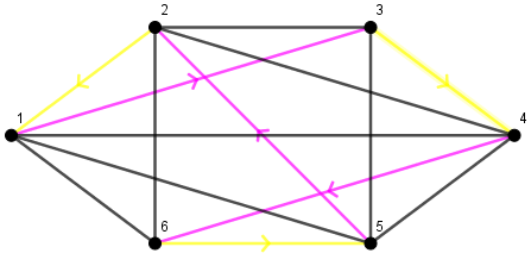

Figura 4.7: Nueva solución T'=[1 3 4 6 5 2 1]

Sin entrar a explicarlo con tanto detalle, ahora se repetiría el proceso de igual forma desde el paso 1 con $T = T'$.
En el **NIVEL 2** se tiene que $X = [(1, 3), (2, 5)]$, $Y = [(3, 2)]$ y XY =[1 3 2 5], $G^* = 5$ y $k = 2$. Al escoger y_2, el valor de $G_2 = 5 = G^*$ por lo que se escoge y_2^* en vez de y_2 .
Los conjuntos de aristas quedarán entonces $X = [(1, 3), (2, 5)]$ e $Y = [(3, 2), (5, 1)]$.
Se genera, por tanto, un intercambio $2 - opt$ obteniendose una nueva solución T' con $f(T') = f(T) - G^* = 22$ representada en el grafo siguiente:

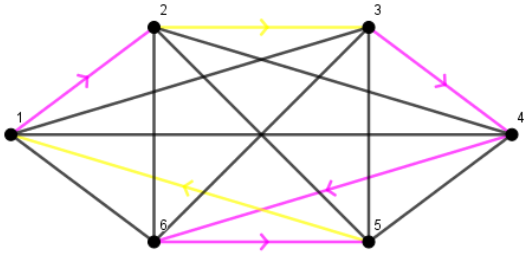

Figura 4.8: Nueva solución T'=[1 2 3 4 6 5 1]

De nuevo, tomando $T = T'$ se repite el proceso. En este nuevo T comenzando por el vértice, 1 como se hizo hasta ahora, la elección de x_1 está limitada, ya que si se escoge la arista $x_1 = (1, 2)$, $|x_1| = 1$ y, por tanto, $g_1 \leq 0$ luego, $x_1 = (1, 5)$. Se sigue el proceso hasta que en el **NIVEL 2** no es posible encontrar un y_2 que cumpla las condiciones. Como $G^* = 0$ se realiza una técnica de backtracking. Se escoge una nueva arista x_2(permitiendo que no cierre ciclo la arista (t_4, t_1)) hasta conseguir después de realizar backtracking en repetidas ocasiones la solución final siguiente con $f(T') = 12$:

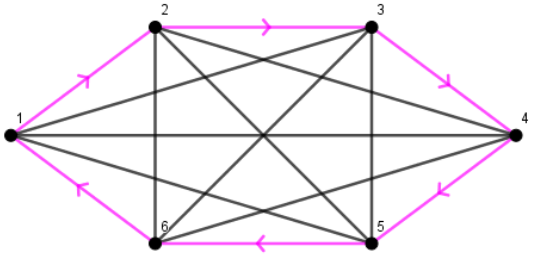

Figura 4.9: Nueva solución T'=[1 2 3 4 5 6 1]

4.5. Pruebas en Matlab con implementación de Lin-Kernighan

En esta sección se hace un pequeño experimento para estudiar la eficiencia del método.

Primeramente, se hace un pequeño estudio para determinar cuál es el valor de k más utilizado para tratar de probar si los $2-opt$ y $3-opt$ son los más comunes, como se expone en las referencias.

Se crean 100 matrices de costes simétricas aleatorias de tamaño 30×30 y se tomará T solución factible de dos maneras distintas; creada por un método constructivo y de manera aleatoria. Teniendo en cuenta esto, se obtienen los resultados:

Con T aleatorio:

- $2 - opt$: 6 veces
- $3 - opt$: 50 veces
- $4 - opt$: 36 veces
- $5 - opt$: 5 veces
- $6 - opt$: 3 veces

Con T de vecino más cercano:

- $2 - opt$: 40 veces
- $3 - opt$: 33 veces
- $4 - opt$: 20 veces
- $5 - opt$: 5 veces
- $6 - opt$: 2 veces

A vistas de esto, se puede ver que la afirmación es cierta con el vecino más cercano, sin embargo, escogiendo un T aleatorio son más comunes los intercambios $4 - opt$ que los $2 - opt$.

Ahora, para determinar la eficiencia y comprobar si una solución es "buena" o no, se empleará la fórmula de a continuación:

$$\text{brecha } (\%) = \frac{(coste\_BB - coste\_LK)}{coste\_BB} \cdot 100$$

de modo que cuanto mejor sea este valor, mejor será la solución obtenida por el método heurístico. de modo que cuanto menor sea mejor será la solución.

Se hizo de nuevo una estudio con 10 matrices simétricas 30×30 determinadas de manera aletario de modo que se aplicó el método branch and bound para poder comparar costes y tiempos de ejecución con Lin Kernighan. Para ello se escogió la solución factible T de dos maneras distintas; mediante la técnica del vecino más cercano y creándola de manera aleatoria. El algoritmo de Lin Kernighan se ejecuta de manera itera mientras la brecha no supere 5 % y no supere el tiempo de ejecucción de Branch and Bound. Los resultados obtenidos se muestran a continuación, donde la primera tabla muestra los resultados y la segunda con una solución aleatoria del vecino más cercano:

Para ambos casos se puede ver que LK tiene un bajo tiempo de ejucción frente al de BB y la brecha es inferior al 5 % y por tanto, bastante buena.
Las diferencias entre los modos de escoger T es notable pero aún así se obtienen muy buenas soluciones.

Número Matriz	Coste inicial	CosteLK	TiempoLK	CosteBB	TiempoBB	Brecha
1	1087	1045	1.1891561	996	6.1940975	4.9196
2	1187	1163	0.2038167	1111	22.7431762	4.68046
3	1133	1046	0.2945601	1014	8.0927479	3.1558
4	1229	1140	0.8224327	1087	5.6769973	4.8758
5	1135	1102	1.3933069	1052	14.3041019	4.7528
6	1165	1080	1.3105758	1029	6.2817252	4.95626
7	1012	1012	0.0002579	976	8.0437434	3.6885
8	999	965	0.0851515	926	9.2128799	4.2116
9	1116	1048	0.9280645	1011	10.0756143	3.6597
10	1218	1218	0.0002428	1162	28.8920664	4.819

Número Matriz	Coste inicial	CosteLK	TiempoLK	CosteBB	TiempoBB	Brecha
1	1087	1045	1.189156	996	6.194097	4.919679
2	1187	1163	0.203817	1111	22.743176	4.680468
3	1133	1046	0.294560	1014	8.092748	3.155819
4	1229	1140	0.822433	1087	5.676997	4.875805
5	1135	1102	1.393307	1052	14.304102	4.752852
6	1165	1080	1.310576	1029	6.281725	4.956268
7	1012	1012	0.000258	976	8.043743	3.688525
8	999	965	0.085152	926	9.212880	4.211663
9	1116	1048	0.928064	1011	10.075614	3.659743
10	1218	1218	0.000243	1162	28.892066	4.819277

Capítulo 5

Aplicación y conclusiones

En esta sección se detallan dos problemas TSP a *gran escala* que se resolverán mediante la implementación en Matlab de los algoritmos presentados en los capítulos anteriores.
Para comprobar que los métodos implementados funcionaban adecuadamente y, por tanto, que daban resultados correctos además de estos dos ejemplos que se muestran a continuación, se utilizaron las instancias del TSP simétrico de la librería de TSP [22] que almacena los costes de las mejores soluciones hasta el momento y las matrices de costes asociadas.

Se mostrarán los resultados del análisis del siguiente modo:
Se obtendrá la solución óptima de los problemas junto a sus costes y tiempos de ejecución mediante el solver ya implementado en Matlab *intlinprog* que se trata de un resolvedor de problemas de Programación lineal de enteros mixtos (MILP). Se introducirá la función objetivo y las restricciones del problema planteado en la sección 2.3.
Primeramente, se obtendrá una solución sin tener en cuenta la restricción (4) para clarificar que esta condición es necesaria. A posteri se añadirá para comprobar su suficiencia.

El objetivo de la utilización de esta función es tener una referencia con la que poder comparar como de *buenas* son las soluciones obtenidas por los otros métodos.

A continuación se expondrán los resultados de la implementación propia de los métodos.

La implementación del método de Branch and Bound se hizo con backtracking por lo que se sabe de antemano que aunque el tiempo de ejecución sea mayor se obtendrá la solución óptima.

El método de Lin Kernighan aunque sea heurístico, se comprobó su eficiencia y gran aproximación a la solución óptima en 4.5.

5.1. Ejemplo práctico TSP Euclídeo

El primero de los problemas planteados considera 38 ciudades situadas en el mapa de Galicia distribuidas según el mapa siguiente:

Figura 5.1: Mapa de Galicia con las ciudades escogidas

Para este caso primer ejemplo, no se tendrá en cuenta la red de carreteras, sino que se va a suponer que todas las ubicaciones están conectadas entre sí por una arista. Se trabajará, por tanto, desde la perspectiva del TSP Euclídeo.

Definición 5.1.1. *Sea $G = (V, A)$ un grafo completo con $|V| = n$ y $C = [c_{ij}]$ una matriz simétrica, se llama* ***TSP métrico*** *si se cumple la desigualdad triangular $c_{ik} \leq c_{ij} + c_{jk}$ $\forall i, j, k \leq n$.*

Definición 5.1.2. *Se llama* ***TSP Euclídeo*** *al problema que considera n puntos en $\mathbb{R}^2$ con distancia euclídea, es decir, $c_{ij} = d(x, y) = \|x - y\|_2$ donde (x, y) son las coordenadas geográficas $(latitud, longitud)$ y se debe encontrar el ciclo hamiltoniano con peso mínimo, es decir, $\sum_{(x,y)\in T} \|x - y\|_2$ sea lo mínimo posible. El TSP Euclídeo es simétrico y métrico.*

Supongamos entonces que nos interesa recorrer un camino que una todas estas ciudades comenzando y acabando en el mismo lugar, de modo que se minimice la distancia recorrida.

Para la construcción de la matriz de costes, es necesario el peso de las aristas y, por tanto, distancias entre las ciudades se ha calculado a partir de las coordenadas geográficas (x, y) =(latitud, longitud) mediante la Fórmula de Haversine.[1]

La fórmula de Harversine es la siguiente:

$$a = sin^2(\tfrac{\varphi_2 - \varphi_1}{2}) + cos(\varphi_1) \cdot cos(\varphi_2) \cdot sin^2(\tfrac{\lambda_2 - \lambda_1}{2})$$
$$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a})$$
$$d = R \cdot c \tag{5.1}$$

Siendo la terminología empleada:

- R: radio de la Tierra (6.371 km)
- φ_1 y φ_2: latitudes en notación decimal
- λ_1 y λ_2: longitudes en notación decimal

Teniendo en cuenta todo esto y siguiente el esquema descrito en el comienzo de esta sección:

5.1.1. Solver *intlinprog*

Para este caso se determinan las soluciones al problema desde las dos perspectivas citadas. No teniendo en cuenta la restricción de subciclos se obtendría lo siguiente:

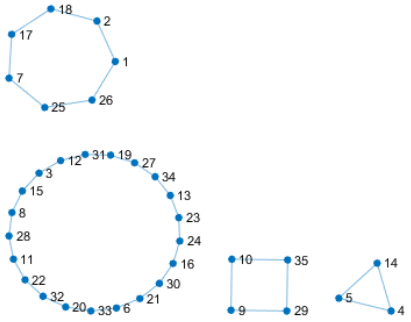

Figura 5.2: Subciclos que se pueden formar sobre el grafo

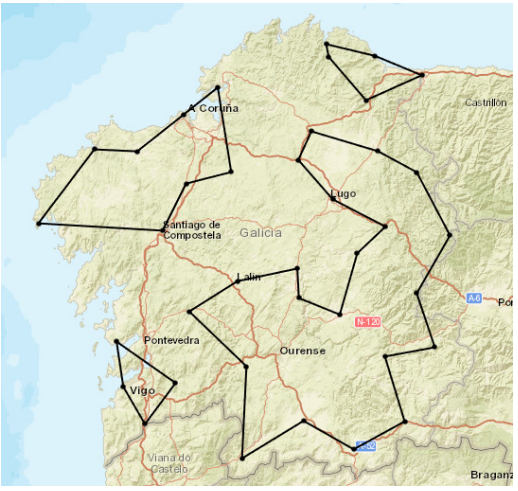

Figura 5.3: Subciclos sobre el mapa

Sin embargo, una vez añadida la restricción se obtiene el ciclo hamiltoniano representado:

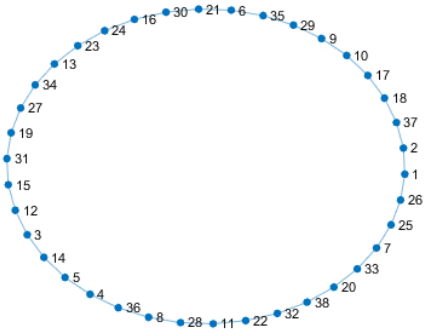

Figura 5.4: Solución óptima en grafo

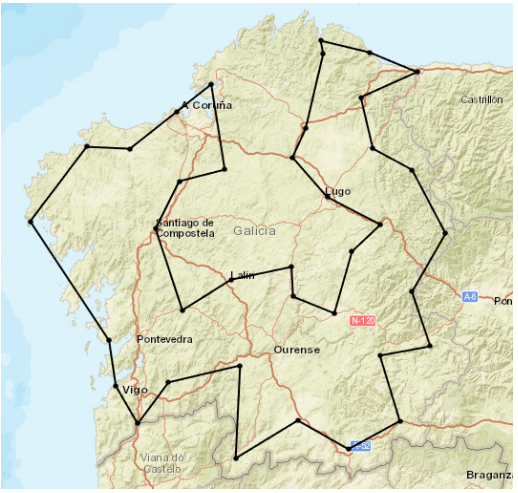

Figura 5.5: Solución sobre mapa

5.1.2. Branch and Bound

Teniendo en cuenta lo discutido en la Sección 3 así como su implementación, se hará una primera búsqueda sin backtracking y a continuación se volverá sobre los nodos vivos para ramificar desde ellos.

La primera solución sin aplicar backtracking con coste 1.1827e+03 y tiempo de ejecución 0,0628 segundos es la siguiente:

33	25	17	37	2	4	8	32	11	19	15	14	12	34	13	23	24	16	21
20	7	18	1	26	36	38	22	28	31	5	3	27	27	23	24	30	6	35

Aplicando backtracking mejora considerablemente el coste obteniendo así la solución óptima:

13	34	27	19	31	15	12	3	14	5	4	36	8	28	11	22	32	38	20
33	7	25	26	1	2	37	18	17	10	9	29	35	6	21	30	16	24	23

5.1.3. Vecino más cercano

El método heurístico voraz da un coste de solución igual a 1.1949e+03 y la solución obtenida es:

1	2	37	18	17	7	25	26	36	4	15	3	5	14	12	31	19	28	11
8	38	32	22	23	13	34	27	24	16	30	21	6	35	29	9	10	33	20

5.1.4. Lin-Kernighan

En este caso, como se detalló en la sección de este método, se empleará el método constructivo del vecino más cercano para tratar de mejorar la solución obtenida mediante una serie de $k - opt$. Se obtiene como solución el ciclo que se muestra a continuación cuyo coste es $1{,}0760e+03$ difiriendo tan solo un 1,3374 % de la solución y con un tiempo de ejecución de 11,6952.

32	20	33	7	25	26	1	2	37	18	17	10	9	29	35	6	21	30
16	24	13	23	22	34	27	19	31	15	12	3	14	5	4	36	8	28

En la siguiente tabla se compactan todos los resultados de coste y tiempo de ejecución para comparar todas las técnicas a la vez:

- L1: intlinprog con tres restricciones
- L2: intlinprog del problema completo
- VC: Vecino más cercano
- BB: Branch and Bound
- LK: Lin Kernighan

	L1	L2	VC	BB	LK
Coste	1.049e+03	1.0618e+03	1.1948e+03	1.0618e+03	1.0760e+03
Tiempo de ejecución(seg)	0.8095	0.5628	0.0083	34.9388	11.6952

Tabla 5.1: Resultados de Coste y Tiempo de ejecución

5.2. Ejemplo práctico TSP

Este segundo ejemplo tratará de una manera más "realista" el problema. El grafo estará constituido por 18 nodos, 18 ciudades situadas en el mapa de Galicia conectadas entre sí por carretera.

Vértice	Ubicación
1	Ferrol
2	A Coruña
3	Castrelo de Miño
4	Santiago de Compostela
5	Lugo
6	Ribadeo
7	Viveiro
8	Finisterre
9	Puebla de Trives
10	Tui
11	Villalba
12	Monforte de Lemos
13	Betanzos
14	Lalín
15	Guitiriz
16	Arzúa
17	Curtis
18	Cerceda

Supongamos que de nuevo nos interesa recorrer un camino que una todas estas ciudades comenzando y acabando en el mismo lugar, de modo que se minimice los kilómetros recorridos.

Se tratará, por tanto, de un grafo ponderado donde las aristas serán las propias carreteras y los pesos de cada una de ellas se corresponden con las distancias en km tomando de referencia la red de carreteras del IGN y con la ayuda de Google Maps [17] escogiendo de ser posible la de menor kilometraje, sin peajes y por autopista.
La ubicación de las ciudades en el mapa es la siguiente:

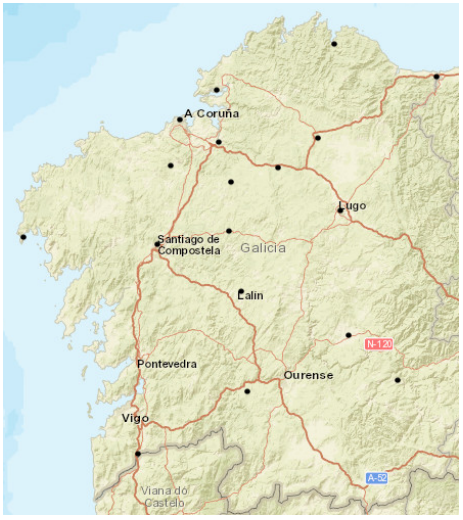

Figura 5.6: Mapa de Galicia con las ciudades escogidas creada con Matlab

Aunque las carreteras no unan ciudades en línea recta, se simplifica el problema para representarlo en un grafo de esta manera. Las aristas de este problema en caso de ser un grafo completo serían $\frac{18*17}{2} = 153$ sin embargo, la figuras siguientes muestran las aristas a tener en cuenta:

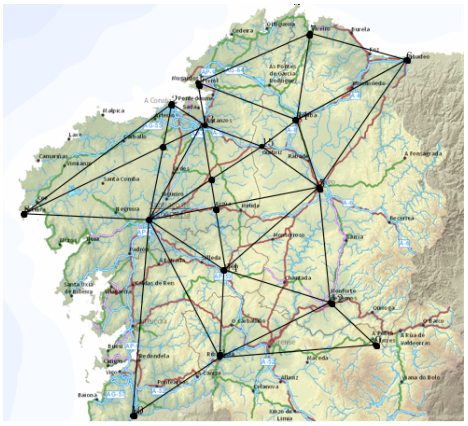

Figura 5.7: Mapa con 18 vértices y aristas existentes

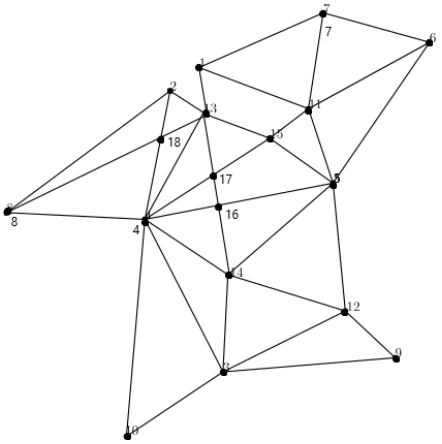

Figura 5.8: Grafo con 18 vértices y aristas existentes

Para la implementación, las aristas que no existen tomarán un valor muy elevado (máximo valor ·1000) para que no sean escogidas. Es por esto que técnicas con la del vecino más cercano no tiene sentido aplicarla en este caso.

5.2.1. Solver *intlinprog*

Primeramente, como se dijo anteriormente, se aplica *intlinprog* y se obtienen las soluciones representadas a continuación.

En caso de no tener en cuenta la restricción de ruptura de subciclos se obtendría la misma que añadiéndola, ya que no se forman ciclos disjuntos de menor longitud en la solución. El coste de la solución óptima representada es 1015.

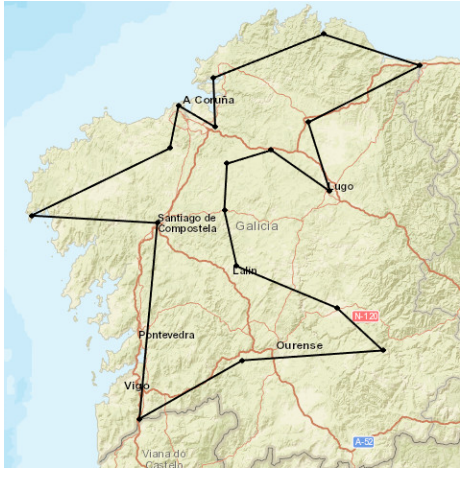

Figura 5.9: Subciclos que se pueden formar en un grafo

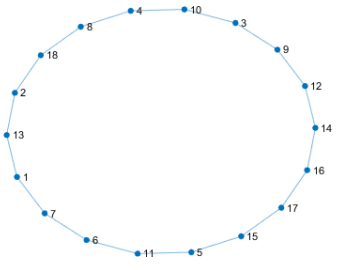

Figura 5.10: Subciclos que se pueden formar sobre el mapa

5.2.2. Branch and Bound

Al igual que para el ejemplo anterior, primero se aplicó el algoritmo sin retroceder en la búsqueda y a continuación se estudiaron los nodos vivos sin utilizar.

Tras haber aplicado backtracking el coste de la solución que se muestra a continuación, disminuye considerablemente con un tiempo de 0,7573 segundos a diferencia de los 0,0612 segundos del paso anterior.

6	7	1	13	2	18	8	4	10
3	9	12	14	16	17	15	5	11

5.2.3. Lin-Kernighan

Para este caso, como el coste de una solución aleatoria o tras aplicar la técnica del vecino más cercano es muy elevado, se tratará de mejorar el "branching a derecha" de Branch and Bound, con este método. Con esto, se obtiene la solución siguiente:

6	11	15	5	14	12	9	3	10
4	8	18	2	17	16	13	1	7

Esta solución es óptima, pues su coste coincide con el determinado al principio de esta sección. El tiempo de ejecución es demasiado elevado para ser esta técnica, pero se debe a que aproximo del todo la solución hasta hacerla óptima.

En la siguiente tabla se mostrarán una recopilación con los costes de las soluciones óptimas obtenidas, así como los tiempos de ejecución, donde la nomenclatura es la siguiente:

- L1: intlinprog con tres restricciones
- L2: intlinprog del problema completo
- BB: Branch and Bound
- LK: Lin Kernighan

	L1	L2	BB	LK
Coste	1015	-	1015	1015
Tiempo de ejecución(seg)	0.1203	-	0.1416	1.1822

Tabla 5.3: Resultados de Coste y Tiempo de ejecución

En conclusión para ambos ejemplos, como se expuso en los capítulo correspondiente, el método heurístico Lin Kernighan da muy buenos resultados en comparación con la solución óptima a pesar de ser muy modificado por su "baja eficiencia".

Bibliografía

[1] *¿Cómo calcular la distancia entre dos puntos geográficos?(Fórmula de Haversine)*. URL: https://www.genbeta.com/desarrollo/como-calcular-la-distancia-entre-dos-puntos-geograficos-en-c-formula-de-haversine.

[2] José Vicente Álvarez. *TEMA 5:COMPLEJIDAD ALGORÍTMICA*. URL: https://www2.infor.uva.es/~jvalvarez/docencia/tema5.pdf.

[3] Clay Mathematics Institute. *P vs. NP*. URL: https://www.claymath.org/millennium/p-vs-np/.

[4] Ismael Crehuet Lucas. «El problema del viajante con grafos». Universidad de Valladolid. URL: https://uvadoc.uva.es/bitstream/handle/10324/57985/TFG-G5977.pdf?sequence=1.

[5] *Definición de Grafos*. URL: https://ccia.ugr.es/~jfv/ed1/tedi/cdrom/docs/grafos.htm.

[6] Reinhard Diestel. *Graph Theory*. Graduate Texts in Mathematics 173. Springer-Verlag, 2006.

[7] Mario Fioravanti. *Apuntes de Matemática Discreta*. 2021-2022. URL: https://personales.unican.es/fioravam/MatDiscreta/Mat-Discreta_3ra-parte_nov2021.pdf.

[8] R.Fulkerson y S.Johson G.Dantzing. *Solution of a large scale traveling salesman problem*. 12 April 1954. URL: https://www.rand.org/content/dam/rand/pubs/papers/2014/P510.pdf.

[9] Keld Helsgaun. *An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic*. Inf. téc. Department of Computer Science, Roskilde University, Denmark, 2000. URL: http://www.akira.ruc.dk/~keld/research/LKH/.

[10] Keld Helsgaun. «An effective implementation of the Lin-Kernighan traveling salesman heuristic». En: *European Journal of Operational Research* (2000).

[11] Keld Helsgaun. «General k-opt submoves for the Lin–Kernighan TSP heuristic». En: *Mathematical Programming Computation* 1.2-3 (2009), págs. 119-163.

[12] Miguel Infantes Durán. «Trabajo de Fin de Grado: El Problema del Viajante (TSP)». Tesis de mtría. Universidad de Sevilla (US).

[13] J. Lee. *Imagen puentes de Konigsberg*. 2020. URL: https://www.ksakosmos.com/post/%EA%B7%B8%EB%9E%98%ED%94%84%EC%9D%B4%EB%A1%A0-%EC%84%B8%EC%83%81%EC%9D%84-%EB%B0%94%EA%BE%B8%EB%8B%A4.

[14] S. Lin y B. W. Kernighan. «An Effective Heuristic Algorithm for the Traveling Salesman Problem». En: *Bell Telephone Laboratories, Incorporated, Murray Hill, N.J.* (1971). 15 October 1971.

[15] S. Lin y B. W. Kernighan. «An effective heuristic algorithm for the traveling-salesman problem». En: *Operations Research* 21.2 (1973), págs. 498-516. URL: http://www.jstor.org/stable/169020.

[16] John D. C. Little et al. «An Algorithm for the Traveling Salesman Problem». En: *Operations Research* (6 March 1963).

[17] Google Maps. *Google Maps*. URL: https://www.google.es/maps.

[18] Cecilia Pola Méndez. *Apuntes de Optimización I*. 2022-2023.

[19] *Milestones in the History of the TSP*. 2005. URL: https://www.math.uwaterloo.ca/tsp/history/milestone.html.

[20] Universidad del País Vasco (EHU). *Investigación Operativa. Programación Lineal*. Tema 5. Programación entera. URL: https://ocw.ehu.eus/file.php/19/5._entera.pdf.

[21] *Problema de los Puentes de Königsberg*. URL: https://es.wikipedia.org/wiki/Problema_de_los_puentes_de_K%C3%B6nigsberg.

[22] Gerhard Reinelt. *TSPLIB 95*. 1995.

[23] Alexander Schrijver. *A Course in Combinatorial Optimization*. 1982. URL: https://homepages.cwi.nl/~lex/files/dict.pdf.

[24] *Tema 8:Complejidad Temporal*. URL: https://www.uhu.es/francisco.moreno/gii_mac/docs/Tema_8.pdf#:~:text=Se%20define%20la%20complejidad%20temporal%20de%20Mcomo%20una,M%20para%20ejecutar%20una%20entrada%20de%20longitud%20n.

[25] Dafne Lucía Arias Vilaboa. «Modelos y algoritmos para el problema del viajante. Una aplicación en planificación socio sanitaria». Universidade de Santiago de Compostela (USC), 2021.

Apéndice A

Programas en Matlab

En este apéndice, se incluyen los programas necesarios para el cálculo de soluciones (BB y LK). El resto se adjuntan en un archivo .zip a la memoria.

A. Vecino más cercano

```
function [coste, tour] = vecinomasproximo(matrizcostes)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funcion que implementa el algoritmo de Vecino mas cercano
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%     matrizcostes: Matriz simetrica nxn con los costes entre ciudades
% OUTPUTS:
%        tour: Vector fila con la mejor solucion encontrada
%        coste: Valor del coste de la mejor solucion
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Inicializo la salida y variables necesarias

    n = size(matrizcostes, 1);
    mejor_coste = Inf;
    mejor_tour = [];

    % Intentar iniciar desde cada ciudad
    for inicio = 1:n
        % Inicializacion
        tour = zeros(1, n);
        ciud_visitadas = zeros(1, n);
        ciudad = inicio;
        tour(1) = ciudad;
        ciud_visitadas(ciudad) = 1;
        for i = 2:n
            % Inicializo variables de test de parada
            min_coste = Inf;
            ciudad_mas_cercana = 0;
            for j = 1:n
                % Se debe encontrar la ciudad mas cercana (con menor coste)
                que no fue visitada
                if ciud_visitadas(j)==0 && matrizcostes(ciudad,j)<min_coste
                    min_coste = matrizcostes(ciudad, j);
                    ciudad_mas_cercana = j;
                end
            end

            % Si no se encuentra una ciudad mas cercana, romper el bucle
            if ciudad_mas_cercana == 0
                break;
            end
```

```
44
45              tour(i) = ciudad_mas_cercana;
46              ciud_visitadas(ciudad_mas_cercana) = 1;
47              ciudad = ciudad_mas_cercana;
48          end
49
50          % Verificar si se ha completado un tour
51          if all(ciud_visitadas)
52              coste = calcula_coste(tour, matrizcostes);
53              if coste < mejor_coste
54                  mejor_coste = coste;
55                  mejor_tour = tour;
56              end
57          end
58      end
59      % Si no se encuentra un tour valido, retornar vacio
60      if isempty(mejor_tour)
61          coste = Inf;
62          tour = [];
63      else
64          coste = mejor_coste;
65          tour = mejor_tour;
66      end
67  end
```

B. Branch and Bound

B.1. BB_TSP

```
1   function [mejorsol, costesol, tiempo_branch, tiempo] = BB_TSP(matrizcostes)
2   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
3   % Funci n que aplica el m todo de Branch and Bound hasta conseguir una
4   % soluci n  ptima  y su coste asociado
5   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6   % INPUTS:
7   %       matrizcostes : Matriz sim trica de tama o nxn
8   % OUTPUTS:
9   %       mejorsol      : Vector fila con las aristas que est n incluidas en la
10  %                       soluci n
11  %       costesol      : Valor con el coste de la soluci n
12  %       tiempo_branch : Valor en segundos del ``branching a derecha ''
13  %       tiempo        : Valor en segundos correspondiente al tiempo de ejecuci n
14  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
15      % Inicializamos la salidas y las variables necesarias
16      tic
17      n = size(matrizcostes, 1); % N mero de v rtices que ser n las ciudades
18      % NODO RAIZ
19      % Crear matriz reducida en filas y columnas
20      [matrizred,costered] = reducematriz(matrizcostes);
21      % Coste inferior del nodo ra z y por tanto del problema
22      LB = costered;
23      % RAMIFICACION A DERECHA
24      [LB, ~, Eord, LB_I_lista, matriz_reducida_lista,E_lista,Eord_lista] = BB_base(
        matrizred, LB);
25      Z = LB; % Cota superior
26      tiempo_branch = toc;
27      %SE INICIA BACKTRACKING
28      %Emplear la t cnica de backtraking si fuese necesario
29      i=1;
30      while i < length(matriz_reducida_lista)+1
31          %Eliminar del array los que no cumplan la condicion
32          if LB_I_lista(i)>= Z
33              matriz_reducida_lista(i)=[];
34              LB_I_lista(i)=[];
35              E_lista(i)=[];
36              Eord_lista(i)=[];
37          else
38              i=i+1;
39          end
```

```
    end
    % Se ramifica a derecha en los nodos que es posible
    while ~isempty(matriz_reducida_lista)
        %Encontrar el minimo de los no vacios en LB_I_lista
        [LB_izq, idx_min] = min(LB_I_lista);
        matriz = matriz_reducida_lista{idx_min};
        % Reduzco la matriz para poder ramificar
        [matrizred, ~] = reducematriz(matriz);
        LB = LB_izq;
        % Ramificar sobre esa matriz reducida a derecha
        [LB_temp, E_temp,Eord_temp,LB_I_lista_temp, matriz_reducida_lista_temp,
    E_lista_temp,Eord_lista_temp] = BB_base(matrizred, LB,Z,E_lista{idx_min},
    Eord_lista{idx_min});
        %Se elimina ese nodo vivo de las listas ya que fue ramificado
        matriz_reducida_lista(idx_min)=[];
        LB_I_lista(idx_min)=[];
        E_lista(idx_min)=[];
        Eord_lista(idx_min)=[];
        %Se concatenan las listas
        matriz_reducida_lista = [matriz_reducida_lista,matriz_reducida_lista_temp];
        LB_I_lista = [LB_I_lista,LB_I_lista_temp];
        E_lista=[E_lista,E_lista_temp];
        Eord_lista=[Eord_lista,Eord_lista_temp];
        % Se comprueba si la ramificacion a derecha de ese nodo vivo
        % dio lugar a una soluci n completa y con menor coste,
        % actualizandola si es el caso
        if size(E_temp,1)==n && LB_temp<Z
            Z = LB_temp;
            Eord = Eord_temp;
            i=1;
            % Eliminar de la lista nodos vivos tras la actualizacion
            % del coste
            while i < length(matriz_reducida_lista)+1
                %Eliminar del array los que no cumplan la condicion
                if LB_I_lista(i)>=Z
                    matriz_reducida_lista(i)=[];
                    LB_I_lista(i)=[];
                    E_lista(i)=[];
                    Eord_lista(i)=[];
                else
                    i=i+1;
                end
            end
        end
    end
    costesol = Z;
    mejorsol = Eord{1};
    tiempo = toc;
end
```

B.2. BB_base

```
function [LB, E, Eord, LB_I_lista, matriz_reducida_lista,E_lista,Eord_lista] =
    BB_base(matrizred, LB,Z,E,Eord)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funcion que ramifica sobre los nodos a derecha. En primer lugar, se
% empleara para obtener una solucion factible pero no necesariamente
% optima.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%       matrizred : Matriz simetrica reducida de tama o nxn
%       LB    : Cota inferior de v(P)
%       Z     : Cota superior de v(P)
%       E     : Matriz nx2 que contiene las aristas incluidas en la solucion
%       Eord  : Array que contiene en cada celda una matriz kx2, donde estan
%               concatenadas k aristas
%   NOTA: Si nargin==2 solo son necesarios los dos primeros inputs ya que se
%                       encuentra en el "branching a derecha" del algoritmo
%         Si nargin >2 se tiene una solucion factible y por tanto una cota
%                       superior Z ademss de una solucion parcial con
```

```
%                       aristas incluidas E y dichas aristas ordenadas Eord
%                       (a excepcion del nivel i=1 que estaran vacios)
% OUTPUTS:
%       LB : Cota inferior de la solucion factible obtenida
%       E: Matriz nx2 que contiene las aristas incluidas en la solucion
%       Eord: Array con una celda que contiene las aristas ordenadas de
%             la solucion
%       Listas para almacenar informacion de nodos "a izquierda" de este
%       "branching a la derecha"
%           LB_I_lista: Vector fila que almacena las cotas inferiores
%                       correspondientes a las soluciones parciales de los
%                       nodos de la izquierda
%           matriz_reducida_lista : Array con las matrices ya reducidas de
%                                   los nodos de la izquierda
%           E_lista: Array con las aristas que se van incluyendo para crear
%                    la solucion parcial por tanto, en el orden de inclusion
%           Eord_lista: Array con las aristas de E ordenadas en cadenas
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %Inicializo la salida para almacenar la informacion de los nodos a
    %izquieda
    LB_I_lista = [];
    matriz_reducida_lista = {};
    E_lista={};
    Eord_lista={};

    % Almacenar las aristas que estan incluidas en el ciclo
    if nargin==2 %Todavia no inclui ninguna arista
        E = []; % aristas incluidas
        Eord = {};
    end
    % Bucle iterativo hasta se reduzca la matriz a 2x2 con elementos no inf
    while sum(matrizred(:) ~= Inf) > 2
        % Calcular la penalizacion de los 0 para ver las elecciones posibles
        [max_penalizacion,fila,columna]=calcula_penalizacion(matrizred);
        LB_I = LB + max_penalizacion; %cota inferior nodo izquierda
        %si el nodo a izquierda creado es vivo por tener menor LB se a ade
        if nargin==2 || LB_I <Z
            %Se reduce la matriz del nodo de la izquierda
            matriznueva = matrizred;
            matriznueva(fila, columna) = Inf;
            matriznueva = reducematriz(matriznueva);
            % Se a ade a las listas la informacion
            LB_I_lista = [LB_I_lista, LB_I];
            matriz_reducida_lista{end + 1} = matriznueva;
            E_lista{end+1}=E;
            Eord_lista{end+1}=Eord;
        end
        % A adir la arista seleccionada a la lista de aristas incluidas
        E = [E; fila, columna];
        % Para RAMIFICAR POR LA DERECHA
        % Establecer infinito para la fila y columna seleccionadas  para
        % que no se puedan escoger elementos de estas
        matrizred(fila, :) = Inf;
        matrizred(:, columna) = Inf;
        % A adido manual para la no formacion de ciclos de menor longitud
        matrizred(columna,fila) = Inf; %evitar ciclo con la arista nueva a adida
        [Eord,e1,e2] = verificar_exciclo(Eord,[fila,columna]);
        matrizred(e1,e2) = Inf;
        % Actualizacion de matrizred
        [matrizred,costered] = reducematriz(matrizred);
        % Coste
        LB = LB +costered;
        if nargin > 2 && LB >= Z
            return
        end
    end
    % Finalmente, cuando queden dos valores distintos de infinito, se
    % a aden ambos
    [filas, columnas] = find(matrizred ~= Inf);
    for i = 1:length(filas)
```

```
        E = [E; filas(i), columnas(i)];
        Eord = verificar_exciclo(Eord,[filas(i),columnas(i)]);
        LB = LB+ matrizred(filas(i),columnas(i));
    end
end
```

B.3. reducematriz

```
function [matrizred,costered] = reducematriz(matrizcostes)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que obtiene una matriz reducida en filas y columnas
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%INPUTS:
%       matrizcostes: matriz sim trica de tama o nxn
%OUTPUTS:
%       matrizred   : matriz sim trica reducida de tama o nxn
%       costered    : valor del coste de la reducci n
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %Inicializamos los elementos de salida
    n = size(matrizcostes,1);
    matrizred = matrizcostes;
    %FILAS
    u = min(matrizcostes,[],2);
    for i = 1:n
        % Si toda la fila es infinito, establecer ui en 0
        if all(isinf(matrizcostes(i, :)))
            u(i) = 0;
        else
            matrizred(i, :) = matrizred(i, :) - u(i);
        end
    end
    %COLUMNAS
    v = min(matrizred, [], 1);
    for j = 1:n
        % Si toda la columna es infinito, establecer vj en 0
        if all(isinf(matrizred(:, j)))
            v(j) = 0;
        else
            matrizred(:, j) = matrizred(:, j) - v(j);
        end
    end
    costered = sum(u) + sum(v);
end
```

B.4. calcula_penalización

```
function [max_penalizacion,fila,columna]=calcula_penalizacion(matriz)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que calcula las penalizaciones de la matriz reducida para saber
% que cero escoger y por tanto por que arista ramificar.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%INPUTS:
%       matriz: matriz sim trica reducida de tama o nxn
%OUTPUTS:
%       max_penalizacion: valor de la penalizaci n m xima y por tanto la
%       elegida
%       fila:  ndice  de la fila de la matriz que contiene el 0 con mayor
%              penalizacion
%       columna :  ndice  de la columna de la matriz que contiene el 0 con
%                  mayor penalizacion
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    n = size(matriz,1);
    penalizacion = zeros(n); %Matriz en donde los elementos ~= 0 de la matriz de
    costes tendr n valor 0
                             % y los elementos = 0 la suma del elemento minimo de
    la fila m s el elemento minimo de la columna
    minimo_fila = zeros(n);  % Matriz constru da de igual forma que la
    penalizaci n pero en los elementos = 0 ser  el m nimo
                             % la fila que corresponda
```

```
    minimo_columna = zeros(n); % Matriz constru da de igual forma que la
    penalizaci n pero en los elementos = 0 ser  el minimo
                            % la columna que corresponda

    for i = 1:n  %indice fila
        for j=1:n  %indice columna
            if matriz(i,j)==0
                minimo_fila(i,j) = min(matriz(i, setdiff(1:n,j)));
                minimo_columna(i,j) = min(matriz(setdiff(1:n,i),j));
                penalizacion(i,j)= minimo_fila(i,j)+minimo_columna(i,j);
            end
        end
    end
    [max_penalizacion, ~] = max(penalizacion(:));
    % Indices del 0 con la penalizaci n  m  s alta
    [fila, columna] = find(penalizacion == max_penalizacion);

    % Si el vector fila tiene  m  s de un elemento, buscar el  m ximo de la
    penalizaci n de la fila
    if length(fila) > 1
        % Inicializo para test de parada
        max_minimo_fila_val = -inf;
        mejor_ind = 0;

        for k = 1:length(fila)
            if minimo_fila(fila(k), columna(k)) > max_minimo_fila_val
                max_minimo_fila_val = minimo_fila(fila(k), columna(k));
                mejor_ind = k;
            end
        end
        % Seleccionar la fila y columna correspondientes
        fila = fila(mejor_ind);
        columna = columna(mejor_ind);
    else
        % Seleccionar el  nico  valor encontrado
        fila = fila(1);
        columna = columna(1);
    end
end
```

B.5. verificar_exciclo

```
function [Eord,e1,e2] = verificar_exciclo(Eord,arista_nueva)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que obtiene un array cuyas celdas contienen las cadenas disjuntas
% que forman las aristas a adidas hasta ahora a la soluci n. El objetivo
% es "colocar" la arista a adida en la cadena correspondiente (uniendo dos
% cadenas si se diese el caso) adem s de reconocer los extremos de los
% nuevos ciclos de longitud menor a n que se deben evitar.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%       Eord : Array que contiene las cadenas tras ir 'pegando' las aristas
%              a adidas hasta el momento
%       arista_nueva : Vector 2x1 que se corresponde con la arista que es
%                      a adida a la soluci n en cada paso
% OUTPUTS:
%       Eord : Array que contiene una arista  m  s que el de entrada en donde
%          cada celda es una matriz kx2, donde est n concatenadas k aristas
%       e1 : Valor del v rtice de un extremo del recorrido formado para
%            evitar ciclo
%       e2 : Valor del v rtice del otro extremo del recorrido formado para
%             evitar ciclo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %Se designa cada v rtice de la arista que se acaba de a adir
    i0 = arista_nueva(1);
    j0 = arista_nueva(2);
    %Para la primera iteraci n, se pone el valor de la arista entrante
    if isempty(Eord)
        Eord{1}=[i0,j0];
        e1 = j0;
```

```
        e2= i0 ;
        return;
    end

    %Si es una arista que cierra ciclo hamiltoniano
    if length(Eord)==1
        if j0==Eord{1}(1,1) && i0==Eord{1}(end,2) %MONI cambi  a "end"
            Eord{1}=[Eord{1};[i0,j0]];
            e1 = [];
            e2 = [];
            return
        end
    end

    %Concatenaci n a izquierda de la nueva arista introducida
    for s=1:length(Eord)
        if j0==Eord{s}(1,1)
            Eord{s}=[i0,j0;Eord{s}];

            %Comprobaci n de que no une dos cadenas
            %Concatenaci n a derecha con otro recorrido ya existente
            for r=1:length(Eord)
                if i0==Eord{r}(end,2)
                    Eaux = [Eord{r};Eord{s}];
                    Eord{s} = Eaux;
                    Eord(r) = [];
                    e1 = Eaux(end,2);
                    e2 = Eaux(1,1);
                    return;
                end
            end
            e1 = Eord{s}(end,2);
            e2 = i0;
            return;
        end
    end

    %Concatenaci n a derecha de la nueva arista introducida
    for k=1:length(Eord)
        if i0==Eord{k}(end,2)
            Eord{k}=[Eord{k};i0,j0];
            e1 = j0;
            e2 = Eord{k}(1,1);
           return;
        end
    end
    %Si no concaten  en ning n lado
    Eord{length(Eord)+1}=[i0,j0];
    e1 = j0;
    e2 = i0;
end
```

C. Lin-Kernighan

C.1. LK_TSP

```
function [T_final,coste_inicial,coste_final] = LK_TSP(matrizcostes)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que aplica de manera iterativa LK_base hasta conseguir la mejor
% soluci n
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%    matrizcostes : Matriz sim trica nxn que tiene los costes de ir de
%                       una ciudad a otra
% OUTPUTS:
%     T_final      : Vector fila con la mejor soluci n T' encontrada con
%                      este m todo
%   coste_inicial  :  Valor del coste de la soluci n T
%   coste_final    :  Valor del coste de la soluci n final T_final
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Inicializo la salida
    [coste_inicial, T] = vecinomasproximo(matrizcostes);

    [T_nuevo,G_est,k,Xopt,Yopt] = LK_base(T, matrizcostes);
    if G_est == 0
        T_final = T;
        coste_final = calcula_coste(T_final,matrizcostes);
    else
        while G_est > 0
            %toc
            T = T_nuevo;
            [T_nuevo,G_est,k,Xopt,Yopt] = LK_base(T, matrizcostes);
        end
        T_final = T_nuevo;
        coste_final = calcula_coste(T_final,matrizcostes);
    end
end
```

C.2. LK_base

```
function [T_nuevo,G_est,k,Xopt,Yopt] = LK_base(T, matrizcostes)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que realiza un movimiento k-opt sobre T soluci n factible
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%       T: Vector fila que es soluci n factible del problema, contiene los
%         v rtices en el orden en que se visitan
%       matrizcostes : Matriz sim trica nxn que tiene los costes de ir de
%                       una ciudad a otra
% OUTPUTS:
%       T_nuevo: Vector fila con la soluci n T mejorada tras haber
%                   intercambiado k aristas
%       G_est: Valor de la mejora ganancia encontrada
%       k    : Valor del n mero de aristas intercambiadas
%       Xopt : Matriz kx2 que contiene las aristas a eliminar de T
%       Yopt : Matriz kx2 que contiene las aristas a a adir de T
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Inicialico las variables necesarias y de salida
    n = length(T);
    E = soluc_aristas(T); % Matriz nx2 que contiene las aristas del tour T
    G_est = 0; % Mejor ganancia hasta el momento
    k=0;
    Xopt=[];
    Yopt=[];
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %   Creamos variables M_X y M_Y
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Crear una matriz en donde solo aparezcan los costes de los elementos que
    % est n en T y el resto 0 (cuando tenga que escoger a partir de x4 el m x
    % que no tenga en cuenta los inf)
    M_X = zeros(n);
    for i = 1:n-1
        M_X(T(i), T(i+1)) = matrizcostes(T(i), T(i+1));
        M_X(T(i+1), T(i)) = matrizcostes(T(i+1), T(i));
    end
    M_X(T(n), T(1)) = matrizcostes(T(n), T(1));
    M_X(T(1), T(n)) = matrizcostes(T(1), T(n));

    % Crear una matriz en donde solo aparezcan los costes de los elementos que
    % NO est n en T (y sus inversas) y el resto inf
    M_Y = matrizcostes;
    for i = 1:n-1
        M_Y(T(i), T(i+1)) = inf;  M_Y(T(i+1), T(i)) = inf;
    end
    M_Y(T(n), T(1)) = inf;  M_Y(T(1), T(n)) = inf;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Empiezo el proceso de LK
```

```
    back_1 = 1:n; %Creo una lista con los v rtices existentes para t1
    while ~isempty(back_1)
        %%%%%%%%%%%%%%%%%%%%%%%%   NIVEL 1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% t1
        % Escoger t1 aleatoriamente
        t1 = back_1(randi(length(back_1))); % Escoger t1 aleatorio
        back_1 = setdiff(back_1, t1); % Eliminar t1 de la lista de opciones a
    explorar
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%% t2
        % Obtener los valores adyacentes a t1 en V, dos opciones para t2
        back_2 = find(M_X(t1,:) ~= 0);
        while ~isempty(back_2)
            % Obtener los valores adyacentes a t1 en V, dos opciones para t2
            %Escoger un t2 de manera aleatoria
            t2 = back_2(randi(length(back_2)));
            back_2 =  setdiff(back_2, t2); %guardo la otra opci n para el
    backtraking
            X = [t1,t2];
            contador_3 = 0;
            %%%%%%%%%%%%%%%%%%%%%%%%%%%% t3
            % Buscar y1 que cumpla las condiciones:
            % 1. No este en T y no sea de los v rtices ya usados en X
            % 2. Tenga peso m nimo
            aux=[1:n;M_Y(t2,:)];
            aux=aux(:,setdiff(1:n,X));
            infs=isinf(aux(2,:));
            aux(:,infs)=[];
            [~,ord_t3] = sort(aux(2,:));
            back_3=aux(1,ord_t3);
            while  ~isempty(back_3) && contador_3 < 5
                t3 = back_3(1);
                contador_3 = contador_3 + 1;
                %Eliminar la opci n ya utilizada
                back_3(1)=[];
                ganancia = M_X(t1,t2)-M_Y(t2,t3); % G1 = g1
                if ganancia<1
                    break %vuelve a buscar otro t2 porque no genera ganancia
                end
                Y = [t2,t3];
                XY = [t1,t2,t3]; %ciclo alternado
                %%%%%%%%%%%%%%%%%%%% NIVEL 2  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                %%%%%%%%%%%%%%%%%%%%%%%%%%%% t4
                %Opciones v lidas para t4
                opt_t4 = find(M_X(XY(end),:) ~= 0);
                % Filtrar las condiciones que tiene que cumplir t4 para
                % Escojo t4 de modo que cumpla las siguientes condiciones:
                % 1. M_X(t4, t1) == 0
                % 2. t4 no es un v rtice que ya est  en XY
                valid_opts = opt_t4(M_X(opt_t4, XY(1)) == 0);
                back_4 = valid_opts(~ismember(valid_opts, XY));
                while  ~isempty(back_4)
                    % Declaro las variables para entrar en caso de que
                    % fuesen modificadas
                    X=[t1,t2];
                    Y=[t2,t3];
                    XY=[t1,t2,t3];
                    [T_2,X,XY,back_4,bool]= encontrar_t4(X,Y,XY,E,M_X,M_Y,back_4);
                    t4 = XY(end);
                    % Solo si bool==1 (en caso de backtracking)
                    if bool==1
                        ganancia_est = M_X(t3,t4)-M_Y(t4,t1); % x1y1x2y2*
    ganancia_2_est
                        ganancia_temp = ganancia + ganancia_est;
                        if ganancia_temp > G_est % T_2 nueva candidata
                            G_est = ganancia_temp;
                            T_prov = T_2;
                            Xopt=[t1,t2;t3,t4];
                            Yopt=[t2,t3;t4,t1];
                            k=2;
                        end
```

```
                    end
                    contador_5 = 0;
                    %%%%%%%%%%%% t5 y t6
                    aux=[1:n;M_Y(t4,:)];
                    aux=aux(:,setdiff(1:n,X));
                    infs=isinf(aux(2,:));
                    aux(:,infs)=[];
                    [~,ord_t5] = sort(aux(2,:));
                    back_5=aux(1,ord_t5);
                    while  ~isempty(back_5) && contador_5 < 5
                        % Declaro las variables para entrar en caso de que
                        % fuesen modificadas
                        X=[t1,t2;t3,t4];
                        Y=[t2,t3];
                        XY=[t1,t2,t3,t4];
                        [t6,sol_ahead,Y,XY,back_5,contador_5]= encontrar_t5(X,Y,XY,
    E,M_X,M_Y,contador_5,back_5);
                        if isempty(t6) %Si no encontr  t5 que crease t6
                            break; %busco otro t4 porque prob  entre todas
                                    %opciones de t5 para ese t4 (puede no
                                    %cerrar ciclo
                        end
                        t5 = XY(end);
                        ganancia = ganancia + (M_X(t3,t4)-M_Y(t4,t5)); % G2 = g1+g2

                        % Ganancia de cerrar con t6
                        ganancia_est = M_X(t5,t6)-M_Y(t6,t1); %x1y1x2y2x3y3*
                        ganancia_temp = ganancia + ganancia_est;
                        if ganancia_temp > G_est % sol_ahead nueva candidata
                            G_est = ganancia_temp;
                            T_prov = sol_ahead;
                            k=3;
                            Xopt=[t1,t2;t3,t4;t5,t6];
                            Yopt=[t2,t3;t4,t5;t6,t1];
                        end
                        if ganancia <= G_est
                            if G_est >0 % Salgo con intercambio 2-opt o 3-opt
                                T_nuevo = T_prov;
                                return
                            else %pruebo con otro t5
                                continue
                            end
                        end
                        %%%%%%%%%%%%%%%%%% NIVEL 3 y SUCESIVOS %%%%%%%%%%
                        seguir = 1; % Mientras se pueda seguir iterando
                        txi = t6;
                       while seguir
                            [X,Y,XY,T_i,txi] = aumentar_camino(T,M_X,M_Y,X,Y,XY,txi
    );
                            s=size(X,1);
                            if ~isempty(txi)
                            ganancia = ganancia +(M_X(X(end,1),X(end,2))-M_Y(Y(end
    ,1),Y(end,2))); % G_i = G_{i-1}+gi

                                ganancia_est = M_X(Y(end,2),txi)-M_Y(txi,t1); %
    x1y1x2y2...xiyi*
                                ganancia_temp = ganancia + ganancia_est;
                                if ganancia_temp > G_est % sol_ahead nueva candidata
                                 G_est = ganancia_temp;
                                 T_prov = T_i;
                                 Xopt=[X;Y(end,2),txi];
                                 Yopt=[Y;txi,t1];
                                 k=s+1;
                                end
                                 if ganancia <= G_est
                                     if G_est >0
                                         T_nuevo = T_prov;
                                         k;
%                                        end
                                        seguir=0;
```

```
                                        return
                                    else %hago backtracking a t5
                                        break
                                    end
                                end
                            else % Me quedar a con el  ltimo  intercambio porque no se pudo hacer m s
                                if G_est >0
                                    T_nuevo = T_prov;
                                    return
                                else %bactraking
                                end
                                seguir=0;
                            end % txi
                        end %seguir
                    end %back_5
                end %back_4
            end  %back_3
        end %back_2
    end %back_1
     if isempty(back_1)
         Xopt = [];
         Yopt = [];
         T_nuevo = T;
         k=0;
     end
end
```

C.3. encontrar_t4

```
function [T_nuevo,X,XY,back_4,bool_prov] = encontrar_t4(X,Y,XY,E,M_X,M_Y,back_4)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que determina el v rtice t4  de x2 de modo que pueda cerrar ciclo
% en caso de ser necesario
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%     X         : Matriz kx2 que contiene en las filas las aristas a eliminar
%     Y         :  Matriz kx2 que contiene en las filas las aristas a a adir
%     XY        : Vector que representa el ciclo alterno de longitud 2k
%     E         : Matriz nx2 que contiene las aristas existentes en T
%     M_X       : Matriz nxn que contiene los costes de las aristas  que forman
%                 parte de la solucion T. El resto de elementos toman valor 0
%     M_Y       : Matriz nxn que contiene los costes de las aristas que NO
%                 forman parte de la solucion T. El resto de elementos toman valor INF
%    back_4     : Vector 1xi que contiene las posibles opciones validas para
%                  t4
% OUTPUTS:
%    T_nuevo    :  Matriz nx2 que contiene una soluci n valida x1y1x2y2*
%     X         :  Matriz k+1x2 que contiene en las filas las aristas a eliminar
%     XY        :  Vector que representa el ciclo alterno de longitud 2(k+1)
%    back_4     :  Vector 1x(i-1) que contiene las posibles opciones validas para
%                  t4
%   bool_prov   : Variable booleana que identifica si se encontro variable
%                 valida o no
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   bool_prov=0;
   i=1;
   while bool_prov==0 && i<length(back_4)+1
        t4=back_4(i);
        X_temp=[X;XY(end),t4];
        XY_temp=[XY, t4];
        % Se calcula posible Y en caso de ser necesario cerrar ciclo
        y2_est = [t4,XY(1)];
        Y_temp = [Y;y2_est];
        [bool_prov,sol] = es_tour(E,X_temp,Y_temp);
        i=i+1;
   end
   if bool_prov ==1
        T_nuevo=sol(:,1)';
```

```
    else
        %No encuentro t4 que cierre
        T_nuevo = [];
    end
     X = X_temp;
     XY = XY_temp;
     back_4 = setdiff(back_4,t4); % Elimino el que ya probe
end
```

C.4. encontrar_t5

```
function  [t6,sol_ahead,Y,XY,back_5,contador_5]= encontrar_t5(X,Y,XY,E,M_X,M_Y,cont
    ,back_5)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que determina el v rtice t5  de y2 de modo que rompa un x3
% verificando la existencia de y3*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%     X         : Matriz kx2 que contiene en las filas las aristas a eliminar
%     Y         :  Matriz kx2 que contiene en las filas las aristas a a adir
%     XY        : Vector que representa el ciclo alterno de longitud 2k
%     E         : Matriz nx2 que contiene las aristas existentes en T
%     M_X       : Matriz nxn que contiene los costes de las aristas  que forman
%                 parte de la soluci n T. El resto de elementos toman valor 0
%     M_Y       : Matriz nxn que contiene los costes de las aristas que NO
%                 forman parte de la soluci n T. El resto de elementos toman valor
    INF
%    cont       : Valor que cuenta el n mero de t5 que fueron probados
%    back_5     : Vector 1xi que contiene las posibles opciones validas para
%                  t5
% OUTPUTS:
%     t6        : V rtice que determina x4
%   sol_ahead   :  Matriz nx2 que contiene una soluci n  v lida x1y1x2y2x3y3*
%     Y         :  Matriz k+1x2 que contiene en las filas las aristas a a adir
%     XY        : Vector que representa el ciclo alterno de longitud 2(k+1)
%  contador_5   : Valor que cuenta el numero de t5 que fueron
%                 probados(implementado en una unidad si se encontro t5 valido)
%    back_5     : Vector 1x(i-1) que contiene las posibles opciones  v lidas para
%                  t5
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    n = size(M_X,1);
    % Se calcula t5(y2) de modo que rompa la arista x3 para entrar en el bucle con
    un posible t6
    %Buscar y2 que rompa un x3
    t5_encontrado=0;
    contador_5 = cont;
    while ~isempty(back_5) && t5_encontrado==0
        t5=back_5(1);
        back_5(1)=[];
        Y_pos = [Y; XY(end),t5];
        XY_pos = [XY, t5];

        opt_t6 = find(M_X(t5,:) ~= 0);
        % Escojo t6 de modo que cumpla las siguientes condiciones:
        % 1. M_X(t6, t1) == 0
        % 2. t6 no es un v rtice que ya est  en XY
        opt_val = opt_t6(M_X(opt_t6, XY(1)) == 0);
        back_6 = opt_val(~ismember(opt_val, XY_pos));
        if isempty(back_6) % No se encuentran opciones  v lidas para t6 -> otro t5
            continue;
        else
        % Seleccionar un t6 aleatorio de las opciones  v lidas
            for j = 1:length(back_6)
                t6 = back_6(j);
                % Se actualiza X y XY con la selecci n
                X_ahead = [X; XY_pos(end), t6];
                XY_ahead = [XY_pos, t6];
                % Calcular y3_est y verificar el ciclo con es_tour
                y3_est_ahead = [t6, XY(1)];
                Y_temp_ahead = [Y_pos; y3_est_ahead];
```

```
                [bool, sol2] = es_tour(E, X_ahead, Y_temp_ahead);
                if bool==1 %Encontro cierre salgo del for
                    t5_econtrado = 1;
                    break;
                end
            end
            %Sale del for anterior si ha encontrado un bool=1 o si todos eran bool
    =0
           if bool==1
               %Esto significa que encontr  un t6 v lido para un t5
                % Se encontr  un t5 v lido
                contador_5 = cont + 1;
                % Actualizar Y con el par [t4, t5] de menor peso
                Y = Y_pos;
                XY = XY_pos;
                sol_ahead = sol2(:,1)';
                return;
           else %para ese t5 no hay t6 que cierre
               continue;
           end
       end
    end
    if contador_5==cont
        t6=[];
        sol_ahead=[];
    end
end
```

C.5. aumentar_camino

```
function [Xnew,Ynew,XYnew,T_nuevo,txi] = aumentar_camino(T,M_X,M_Y,X,Y,XY,txi)
    %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Funci n que devuelve los conjuntos X,Y de aristas para realizar un k-opt
    %intercambio donde k es el n mero de aristas intercambiadas
    %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %    INPUTS:
    %      T        : Vector fila que contiene la soluci n inicial
    %     M_X       : Matriz nxn que contiene los costes de las aristas  que forman
    %                 parte de la soluci n T. El resto de elementos toman valor 0
    %     M_Y       : Matriz nxn que contiene los costes de las aristas que NO
    %                 forman parte de la soluci n T. El resto de elementos toman
    valor INF
    %     X         : Matriz kx2 que contiene en las filas las aristas a eliminar
    %     Y         :  Matriz kx2 que contiene en las filas las aristas a a adir
    %     XY        : Vector que representa el ciclo alterno de longitud 2k
    %    txi        : V rtice que determina X, calculado en el paso anterior
    %  OUTPUTS:
    %     Xnew          : Matriz (k+1)x2 que contiene en las filas las aristas a
    eliminar
    %     Ynew          : Matriz (k+1)x2 que contiene en las filas las aristas a
    a adir
    %     XYnew         : Vector que representa el ciclo alterno resultante de
    longitud 2(k+1)
    %    T_nuevo        : Array en el que cada celda tiene la soluci n encontrada
    %                     con un intercambio k
    %      txi          : V rtice que determina la siguente arista a introducir
    %                     en X
    %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    seguir=0;
    n = length(T);
    E = soluc_aristas(T);
    %Inicializo las variables de salida
    Xnew=X;
    Ynew=Y;
```

```
    XYnew=XY;

    %Declaro xi con lo calculado en el paso anterior
    X = [X; XY(end),txi];
    XY = [XY txi];
    %%%%%%%%%%%%%%%%%%%%%% CAMBIADO
    % Buscar un yi que rompa un x_{i+1}
    %ind_rest = setdiff(1:n,XY);
    % Ordenar de menor a mayor ind_rest por los pesos M_Y(XY(end), tyi)
    %[~, ind_ord] = sort(M_Y(XY(end), ind_rest));
    %ind_rest_ord_temp = ind_rest(ind_ord);
    %ind_rest_ord = ind_rest_ord_temp(~isinf(M_Y(XY(end), ind_rest_ord_temp))); %Le
     quito los infinitos porque esas aristas est n en E
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    aux=[1:n;M_Y(XY(end),:)];
    aux=aux(:,setdiff(1:n,XY));
    infs=isinf(aux(2,:));
    aux(:,infs)=[];
    [~,ind_ord] = sort(aux(2,:));
    ind_rest_ord=aux(1,ind_ord);

    for i = 1:length(ind_rest_ord)
        tyi = ind_rest_ord(i);
        Y_pos = [Y; XY(end),tyi];
        XY_pos = [XY, tyi];
        % Busco que cumpla la rotura de x_{i+1}
        opt_txi = find(M_X(tyi,:) ~= 0);
        % Escojo txi de modo que cumpla las siguientes condiciones:
        % 1. M_X(txi, t1) == 0
        % 2. txi no es un v rtice que ya est  en XY
        valid_opts = opt_txi(M_X(opt_txi,XY(1)) == 0);
        opt_val = valid_opts(~ismember(valid_opts, XY_pos));
        if isempty(opt_val)
            continue
        end
        % Ordenar txi por peso y tomar hasta dos opciones
        [~, ind] = sort(M_X(tyi, opt_val), 'descend'); %ordena de mayor a menor
        back_i = opt_val(ind);  % Reordenamos opt_val seg n los pesos
        for j = 1:length(back_i)  % Iterar sobre las dos opciones y quedarse con la
     mejor
            txi = back_i(j);
            X_ahead = [X; tyi, txi];
            XY_ahead = [XY_pos, txi];
            yi_est_ahead = [txi, XY(1)];
            Y_temp_ahead = [Y_pos; yi_est_ahead];
            % Verificar el ciclo con es_tour
            [bool, sol_ahead] = es_tour(E, X_ahead, Y_temp_ahead);
            if bool==1
                break
            end
        end
        if bool==1
            Xnew = X;
            Ynew = [Y; XY(end), tyi];
            XYnew = [XY, tyi];
            T_nuevo = sol_ahead(:,1)';
            seguir = 1;
            return;  % Termina la funci n si encuentra una soluci n v lida
         end
    end
    if seguir==0
        txi=[];
        T_nuevo = [];
    end
end
```

C.6. calcula_coste

```
function coste = calcula_coste(T, matrizcostes)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%Funcion que calcula el coste del ciclo completo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%   ciclo: Vector fila que contiene los v rtices en el orden en el que se
%          recorren
%   matrizcostes: Matriz nxn que contiene los costes de las aristas
% OUTPUTS:
%  coste: Valor que representa el coste del ciclo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Inicializo la salida
coste = 0;
n = length(T);
% Calcula el coste total del tour
for i = 1:n-1
    coste = coste + matrizcostes(T(i), T(i+1));
end
coste = coste + matrizcostes(T(n),T(1));
```

C.7. es_tour

```
function [bool,sol] = es_tour(E,X,Y)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que comprueba si se crea un tour al eliminar las aristas X y
% a adir las de Y
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
% E : Matriz 2*nx2 que contiene las aristas(en las dos direcciones posibles)
%    que formar parte del ciclo
% X : Matriz kx2 cuyas filas ser n las aristas a eliminar del ciclo inicial
% Y : Matriz kx2 cuyas filas ser n las aristas a introducir al ciclo
% inicial OUTPUTS: bool : Booleano que representa si se econtro o no un
% tour correcto sol : Ciclo ordenado construido con E\X U Y
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Inicializo la salida
n=size(E,1);

% Comprobaciones previas
% Comprobar que las aristas de Y(o sus inversas) no forman parte del ciclo
for k=1:size(Y,1)
    idx_Y = ismember(E, Y(k,:), 'rows') | ismember(E, flip(Y(k,:)), 'rows');
    E(idx_Y, :) = [];
end
if size(E,1)~=n
    disp('error: Alguna arista de Y esta en E')
    return
end

% Comprobar que las aristas de X(o sus inversas) forman parte del ciclo
for k=1:size(X,1)
    idx_X = ismember(E, X(k,:), 'rows') | ismember(E, flip(X(k,:)), 'rows');
    E(idx_X, :) = [];
end

if size(E,1)~=n-size(X,1)
    disp('error: No todas las aristas de X estan en E')
    return
end
%Comprobaci n de que el intercambio es posible
% Esto relmente solo se cumple si es para cerrar el ciclo(la segunda
% condicion
if all(Y(:,1)==X(:,2)) && all(Y(:,2)==[X(2:end,1);X(1,1)])
else
    disp('Las X e Y no hacen ciclo')
    return
end
%Construcci n del ciclo ordenado con las aristas correspondientes
%intercambiadas
Eord={}; %Array fila donde cada celda contiene una  matriz (k,2) con una cadena
         % tras haber extraido las aristas de X
for i=1:size(E,1)
```

```
        Eord = verificar_exciclo(Eord,E(i,:));
end

% "Pegado" de las aristas de Y a alguna de las cadenas
for j=1:size(Y,1)
    pegado = 0;
    for i = 1:length(Eord)
        if  Y(j,2) == Eord{i}(1, 1) && pegado == 0
            Eord{i} = [Y(j,:); Eord{i}];
            pegado = 1;
        elseif Y(j,2) == Eord{i}(end, 2) && pegado == 0
            Eord{i} = [Eord{i}; flip(Y(j,:))];
            pegado = 1;
        elseif Y(j,1) == Eord{i}(end, 2) && pegado == 0
            Eord{i} = [Eord{i}; Y(j,:)];
            pegado = 1;
        elseif Y(j,1) == Eord{i}(1, 1) && pegado == 0
            Eord{i} = [flip(Y(j,:)); Eord{i}];
            pegado = 1;
        end
    end
end


%Pegar las posibles cadenas que pueda haber en Eord entre si. Habiendo 4
% posibilidades de pegado
imposible_pegar=0;
while length(Eord)>1 && imposible_pegar==0
    for k=2:length(Eord)
        encontrado=0;
        if Eord{1}(end,2)==Eord{k}(1,1) %  ltima  con primera
            encontrado=1;
        elseif Eord{1}(1,1)==Eord{k}(1,1) % Primera con primera
            Eord{1}=flip(Eord{1});
            Eord{1}=flip(Eord{1},2);
            encontrado=1;
        elseif Eord{1}(1,1)==Eord{k}(end,2) % Primera con  ltima
            Eord{1}=flip(Eord{1}); Eord{1}=flip(Eord{1},2);
            Eord{k}=flip(Eord{k}); Eord{k}=flip(Eord{k},2);
            encontrado=1;
        elseif Eord{1}(size(Eord{1},1),2)==Eord{k}(size(Eord{k},1),2) %  ltima  con
      ltima
            Eord{k}=flip(Eord{k});
            Eord{k}=flip(Eord{k},2);
            encontrado=1;
        end

        if encontrado==1
            Eord{1}=[Eord{1};Eord{k}]; %concatena las dos cadenas
            Eord(k)=[];
            break;
        else
            imposible_pegar=1;
        end
    end
end
% Definir la salida si fue o no posible pegar y por tanto generando una
% unica celda en Eord
if length(Eord)==1
    bool=1;
    sol=Eord{1};
else
    bool=0;
    sol=Eord;
end

end
```

C.8. soluc_aristas

```
function E = soluc_aristas(T)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Funci n que crea a partir de una soluci n representada por los v rtices
% en orden la soluci n con la secuencia de aristas en la soluci n
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INPUTS:
%       T: Vector fila que contiene la soluci n con los v rtices en orden
%        seg n el recorrido
% OUTPUTS:
%       E: Matriz 2nx2 que contiene las aristas de la soluci n y sus
%       inversas
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Inicializamos la salida
    n = length(T);
    E = zeros(n, 2);%MONI
    % Aristas del vector
    for i = 1:n-1
        E(i, 1) = T(i);
        E(i, 2) = T(i+1);
    end
    E(n,1)=T(n); E(n,2)=T(1);
end
```

Apéndice B

Tablas de datos

A. Tabla de coordenadas de las 38 ciudades

Vértice	Ubicación	Latitud	Longitud
1	Ferrol	43,48268	-8,2238
2	A Coruña	43,36762	-8,42287
3	Nois	43,61552	-7,31274
4	Villalba	43,29748	-7,68077
5	Chavín	43,61076	-7,5798
6	Castrelo de Miño	42,29728	-8,05742
7	Santiago de Compostela	42,87686	-8,54417
8	Lugo	43,00973	-7,55675
9	Samil	42,20962	-8,77215
10	Sanxenxo	42,4033	-8,81135
11	Sarria	42,78083	-7,41407
12	Ribadeo	43,53367	-7,04034
13	Comarca de Valdeorras	42,37886	-6,96386
14	Viveiro	43,66426	-7,59453
15	Mondoñedo	43,42857	-7,36384
16	Verín	41,94025	-7,43495
17	Finisterre	42,9078	-9,26503
18	Lage	43,22121	-8,94043
19	Fonsagrada	43,12471	-7,06885
20	Lalín	42,66115	-8,11102
21	Lobios	41,90113	-8,08335
22	Monforte de Lemos	42,51902	-7,5158
23	Puebla de Trives	42,33934	-7,2538
24	A Gudiña	42,06057	-7,1393
25	Ordes	43,07637	-8,40806
26	Curtis	43,12533	-8,14599
27	Piornedo	42,85648	-6,87581
28	Baralla	42,8939	-7,25042
29	Tui	42,04915	-8,6466
30	Xinzo de Limia	42,06311	-7,72594
31	Meira	43,21342	-7,29428
32	Melide	42,58948	-7,75338
33	Cerdedo	42,53219	-8,38949
34	Visuña	42,60963	-7,06917
35	Mondariz	42,22577	-8,46878

36	Baamonde	43,17667	-7,75658
37	Carballo	43,21282	-8,69152
38	Taboada	42,71558	-7,76208

B. Tabla con las distancias entre las aristas existentes de las 18 ciudades

Vértice 1	**Ciudad 1**	**Vértice 2**	**Ciudad 2**	**Distancia en km**
1	Ferrol	7	Viveiro	76
13	Betanzos	1	Ferrol	35
1	Ferrol	11	Villalba	58
2	A Coruña	8	Finisterre	101
13	Betanzos	2	A Coruña	22
4	Santiago de Compostela	8	Finisterre	78
4	Santiago de Compostela	10	Tui	102
4	Santiago de Compostela	13	Betanzos	63
4	Santiago de Compostela	14	Lalín	50
4	Santiago de Compostela	3	Castrelo de Miño	103
3	Castrelo de Miño	10	Tui	70
3	Castrelo de Miño	14	Lalín	55
3	Castrelo de Miño	12	Monforte de Lemos	68
3	Castrelo de Miño	9	Puebla de Trives	93
9	Puebla de Trives	12	Monforte de Lemos	49
12	Monforte de Lemos	14	Lalín	65
12	Monforte de Lemos	5	Lugo	62
5	Lugo	11	Villalba	38
5	Lugo	6	Ribadeo	90
5	Lugo	14	Lalín	73
5	Lugo	15	Guitiriz	42
15	Guitiriz	11	Villalba	30
15	Guitiriz	13	Betanzos	33
6	Ribadeo	7	Viveiro	59
6	Ribadeo	11	Villalba	72
7	Viveiro	11	Villalba	52
16	Arzua	5	Lugo	66
16	Arzua	4	Santiago de Compostela	37
16	Arzua	14	Lalín	39
17	Curtis	15	Guitiriz	29
17	Curtis	13	Betanzos	22
17	Curtis	4	Santiago de Compostela	49
17	Curtis	16	Arzua	27
18	Cerceda	13	Betanzos	36
18	Cerceda	2	A Coruña	26
18	Cerceda	4	Santiago de Compostela	50
18	Cerceda	8	Finisterre	93