Algoritmos para el problema de ruta mínima

Marta Sáenz Diez

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Director del trabajo: Pedro M. Mateo Collazos
12 de junio de 2024

Abstract

Currently, network optimization has become a main field in Operational Research. This area includes problems such as the shortest path and network flows among others. The shortest path problem is considered one of the most significant due to its numerous applications and the existence of efficient methods for complex problems. It consists of finding the most efficient way to get from one point to another, considering the costs associated to each feasible route.

In this context, networks are used to describe and model real-world systems, in particular, directed graphs, being each node an object, place, or action, while the arcs are the connections between them, which have a cost associated. The problem is reduced to finding the minimum distance between two given points.

Network representation is used in a wide variety of circumstances, being, logistics, communication and electrical networks among the most prevalent. It is also used in production, distribution, and management, providing visual support of the relation between every component of the system.

The shortest path problem has not a particularly not complex structure, allowing the development of several intuitively appealing algorithms for its resolution. One of the most remarkable algorithms is the Dijkstra's algorithm, published in 1959 by Edsger Dijkstra. This algorithm and its numerous modifications have abundant applications, but the one that stands out from the rest is the use of GPS. The Bellman-Ford algorithm is another outstanding one, published in 1958. It is more general since, unlike the previous, it does not exclude networks with negative costs.

However, while shortest path problems may seem straightforward to solve, designing and analyzing some of the most efficient algorithms demands significant ingenuity.

Below, the content of this work is described and organized in 5 chapters.

In Chapter 1, an introduction to the shortest path problem is given, in addition to defining the basic graph concepts that will be used. The shortest path problem is defined in detail, employing knowledge from Operational Research. Additionally the existing variants of the problem are outlined, and at the end, two types of algorithms can be distinguished the *label-setting* and *label-correcting*.

Moving on to Chapter 2, the *generic algorithm* is explained in detail, along with its pseudocode. Convergence properties are formulated and demonstrated, and an alternative initialization is presented. Finally, the algorithm is applied to a base example.

In Chapter 3, the label-setting algorithms are explained, which do not allow the existence of negative costs in the graph. The first to be introduced is the Dijkstra's algorithm which is studied along with its main properties, as well as its computational complexity. Further to this, in order to improve its complexity, two algorithms, which are modifications of Dijkstra, are introduced. These are the Dijkstra's algorithm with priority queues, also known as Heap, and the Dial's algorithm, which uses buckets. The pertinent pseudocode is added along with an application in a base example.

In Chapter 4, the label-correcting algorithms are studied. Note that in this case the algorithms are not restricted to graphs with non-negative costs, making them more generic algorithms since negative costs are allowed. First, the Bellman-Ford algorithm is explained, along with its properties, which is is followed by the D'Esopo-Pape and SLF algorithms, being these last ones

modifications of the generic algorithm. All of them are accompanied by their pseudocodes and applications on a base example.

Finally, in Chapter 5, a study for networks with either positive or negative costs is conducted to contrast the performance of all the algorithms studied in the previous chapters. In order to do this, the algorithms are implemented using the programming language *Python*. This study compares the average execution time in milliseconds of each algorithm according to the number of nodes in the network, its density, and shape.

Índice general

Capítulo 1

Introducción

Hoy en día, el problema de ruta mínima está continuamente presente en la vida cotidiana. ¿Cómo ir de un objetivo a otro distinto minimizando lo máximo posible el coste de esta actividad? Esta, es una cuestión que si uno se para a pensar, se da cuenta que realmente se puede aplicar a todo lo que le rodea.

¿Cómo ir de un punto de tu ciudad a otro en el recorrido más corto posible? Tanto *Google Maps* como cualquier otro sistema de navegación son capaces de resolver este problema tan habitual en el que se busca la ruta más corta entre dos puntos del mapa. Pero, para usar este tipo de aplicaciones, o hacer cualquier otra consulta externa, es necesario una conexión a Internet, lo que lleva a investigar como transmitir información entre una o varias redes de Internet en el menor tiempo posible. Estos paquetes de información, buscan la ruta más óptima que hay entre las redes que desean transmitir el mensaje, y lo hacen mediante el *routing*. La mayoría de las veces se trata en efecto de su ruta mínima, que, junto con otros factores, es lo que genera la eficiencia de las conexiones a la red y evita los tiempos de espera en su carga. Sin embargo, no basta con eso, ya que para la carga de los aparatos electrónicos por los que se realice dicha consulta, se tendrá que hacer uso del sistema eléctrico, buscando así distribuir la mayor cantidad de energía posible al menor costo. Para ello se utilizan problemas de ruta mínima, encontrando así la ruta mas corta y el menor costo de construcción de la red eléctrica.

El problema de ruta mínima es uno de los problemas con más relevancia en el ámbito de la optimización, y esto es debido a sus numerosas aplicaciones, de las cuales solo se han dado unos pequeños ejemplos, para ilustrar cómo pueden llegar a aplicarse todos los ámbitos. En concreto, cobran más relevancia en la planificación logística, ya que pueden ayudar a mejorar significativamente la eficiencia operativa, así como reducir los costos a su vez.

Para resolver toda esta variedad de problemas planteados, se harán uso de varios algoritmos, los cuales se especificarán más adelante en el trabajo con su respectiva programación en el lenguaje *Python*.

1.1. Definición del problema de ruta mínima

Para el desarrollo de las definiciones se ha hecho uso de [1], mientras que para el planteamiento del problema se ha ha utilizado [2].

Definición 1. Un *grafo* es un par $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, donde $\mathcal{N} = \{1, \ldots, N\}$ finito y cuyos elementos son denominados nodos y $\mathcal{A} = \{(i, j) | i, j \in \mathcal{N}\} \subseteq \mathcal{N} \times \mathcal{N}$ es un subconjunto de $\mathcal{N} \times \mathcal{N}$ cuyos elementos se denominan arcos. Se distinguen los siguientes tipos:

- *Grafos dirigidos:* el par (i, j) está ordenado y el arco (i, j) será distinto del arco (j, i).
- *Grafos no dirigidos:* sus arcos son no dirigidos, es decir, se tiene que (i, j) y (j, i) representan el mismo arco.

Al número de sus nodos, $|\mathcal{N}| = n$, se le denota como orden del grafo, mientras que el número total de arcos, $|\mathcal{A}| = m$, es el tamaño del grafo $\mathcal{G}$.

Definición 2. Dado un grafo $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ un *camino dirigido* $\mathcal{P}$ de longitud l desde el nodo i al j es una sucesión de nodos $(i_0, i_1, \ldots, i_l)$ tal que $i_0 = i$, $i_l = j$ y $(i_{h-1}, i_h) \in \mathcal{A}$ para $h = 1, \ldots, l$, en donde, además, ninguno de sus arcos aparecen repetidos. En particular, se denomina *camino dirigido simple* cuando en la sucesión de nodos $(i_0, i_1, \ldots, i_l)$ no aparece ninguno repetido.

Todos los ejemplos explicados previamente en la introducción, pueden ser representados a través de grafos, o más concretamente, de grafos dirigidos, que serán los protagonistas en este trabajo. Aunque los algoritmos de resolución de rutas mínimas que se presentarán están basados en grafos dirigidos, pueden ser modificados para aplicarse a grafos no dirigidos. Además, existen algoritmos específicos diseñados para grafos no dirigidos.

A partir de ahora, dado un grafo dirigido $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, definido en los términos anteriores, se considerará que cada uno de sus arcos $(i, j) \in \mathcal{A}$ tiene un costo asociado c_{ij}.

Sea $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ con costos c_{ij} para todo $(i, j) \in \mathcal{A}$. Con esto se define el problema de ruta mínima como aquel que trata de encontrar el camino dirigido simple más corto, $\mathcal{P}$, entre ciertos nodos, que sin pérdida de generalidad, se asumirá que son el 1 y el N, respectivamente. El problema puede formularse inicialmente como:

$$(P)\begin{cases} \text{minimizar} & \displaystyle\sum_{(i,j)\in\mathcal{P}} c_{ij}x_{ij} \\ \\ \text{sujeto a:} & (1 \ldots N) \text{ camino dirigido de 1 a } N \text{ en } \mathcal{A}. \end{cases}$$

Aunque la forma anterior es la manera más intuitiva de definir el problema de ruta mínima, habitualmente este se modela mediante un procedimiento distinto, tal y como se puede ver a continuación.

El problema de ruta mínima se puede definir como el problema que busca la solución óptima del siguiente problema de programación lineal entera (PLE) binaria[1].

$$(\text{P1})\begin{cases} \text{minimizar} \displaystyle\sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \\ \\ \text{sujeto a:} \displaystyle\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = \begin{cases} 1 & \text{si } \ i = 1, \\ 0 & \text{si } \ i \neq 1, N, \\ -1 & \text{si } \ i = N, \end{cases} \qquad i \in \mathcal{N} \\ \\ x_{ij} \in \{0, 1\}, \qquad (i, j) \in \mathcal{A}. \end{cases}$$

Dadas las especiales características del problema, este puede remodelarse, reformulando así las condiciones de las variables x_{ij}, que pasan de ser $x_{ij} = 0$ ó 1 a $x_{ij} \geq 0$ para todo $(i, j) \in \mathcal{A}$. Esto puede afirmarse ya que se trata de se trata de un caso particular de un problema de transporte, por esta razón esta reformulación no añadirá ninguna solución óptima extra. De esta forma, se trata de un problema de programación lineal (PPL) estándar.

El problema de ruta mínima que se considera en adelante viene dado por el sistema (P2).

$$(\text{P2})\begin{cases} \text{minimizar} \displaystyle\sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \\ \\ \text{sujeto a:} \displaystyle\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = \begin{cases} 1 & \text{si } \ i = 1, \\ 0 & \text{si } \ i \neq 1, N, \\ -1 & \text{si } \ i = N, \end{cases} \qquad i \in \mathcal{N} \\ \\ x_{ij} \geq 0, \qquad (i, j) \in \mathcal{A}. \end{cases} \tag{1.1}$$

[1] El modelado de este tipo de problemas se estudió en la asignatura de Investigación Operativa, correspondiente al tercer curso del grado de Matemáticas.

A continuación, se van a utilizar los conocimientos sobre dualidad en programación lineal adquiridos en la asignatura Investigación Operativa cursada en el tercer curso del grado de Matemáticas, de donde se podrán derivar las condiciones de optimalidad para el problema de ruta mínima, de una forma alternativa a como se hace normalmente.

De acuerdo a la teoría de la dualidad para programación lineal, asociado a cada PPL, existe otro problema de programación lineal, denominado problema dual, cuyas soluciones óptimas están totalmente relacionadas. Dada esta teoría, el problema dual asociado a 1.1 será el siguiente.

$$(D)\begin{cases}\text{maximizar} & w_1 - w_m \\ \text{sujeto a:} & w_i - w_j \leq c_{ij}, \qquad (i,j) \in \mathcal{A} \\ w_i \in \mathbb{R}, & i \in \mathcal{N}.\end{cases} \tag{1.2}$$

Este entorno, da lugar a la siguiente definición.

Definición 3. Un *ciclo dirigido* es un camino dirigido cuyos nodos inicial y final coinciden ($i_0 = i_l$). Se denominará *ciclo negativo* si la suma de los costos de los arcos que forman el ciclo es un número negativo.

El siguiente teorema establece las denominadas condiciones de holgura complementaria (CHC), en el que dadas unas soluciones factibles para un problema primal y su dual, estas son óptimas si y solo si al multiplicar el valor de la holgura de la restricción del problema primal (dual) por el valor de la variable correspondiente del problema dual (primal) el resultado es cero.

Teorema 1.1 (Condiciones de Holgura Complementaria[2])**.** *Sea $\bar{\boldsymbol{x}}$ y $\bar{\boldsymbol{w}}$ soluciones factibles de los problemas simétricos de máximo y su dual. Entonces $\bar{\boldsymbol{x}}$ y $\bar{\boldsymbol{w}}$ son soluciones óptimas para sus problemas respectivos si y solo si $\bar{x}_j \bar{v}_j = 0$ para $j = 1, .., n$ y $\bar{w}_i \bar{u}_i = 0$ para $i = 1, .., m$ con $\bar{\boldsymbol{u}}$ y $\bar{\boldsymbol{v}}$ holguras del primal y dual respectivamente.*

El teorema anterior aplicado al problema de ruta mínima toma la forma siguiente.

Teorema 1.2 (Condiciones de Holgura Complementaria para el problema de ruta mínima)**.** *Sea $\bar{\boldsymbol{x}}$ e $\bar{\boldsymbol{w}}$ soluciones factibles de los problemas 1.1 y 1.2 respectivamente. Entonces $\bar{\boldsymbol{x}}$ y $\bar{\boldsymbol{w}}$ son soluciones óptimas para sus problemas respectivos si y solo si se tiene $\bar{x}_{ij}(c_{ij} - \bar{w}_i + \bar{w}_j) = 0$ para $(i,j) \in \mathcal{A}$.*

El teorema 1.2 permite proporcionar las condiciones habituales de optimalidad para el problema de ruta mínima.

Proposición 1.3. *Sea $d = (d_1, d_2, ..., d_N)$ un vector de distancia cumpliendo*

$$d_j \leq d_i + c_{ij}, \quad \text{para todo } (i,j) \in \mathcal{A}, \tag{1.3}$$

y $\mathcal{P}$ un camino que comienza en el nodo 1 y termina en el nodo N. Entonces, si

$$d_j = d_i + c_{ij}, \quad \text{para todo } (i,j) \in \mathcal{P}, \tag{1.4}$$

$\mathcal{P}$ es el camino de ruta mínima desde 1 hasta N.

Demostración. Dado $\mathcal{P}$, un camino que comienza desde el nodo 1 a N, definiendo $x_{ij} = 1$ para todo $(i,j) \in \mathcal{P}$ y 0 para el resto de arcos, será una solución factible del problema primal 1.1. Definiendo $w_i := -d_i$, se cumplirá

$$w_i \leq w_j + c_{ij}, \quad \text{para todo } (i,j) \in \mathcal{A},$$

[2]Los elementos involucrados en dicho teorema se pueden encontrar en el Anexo A, los cuales fueron estudiadas en la asignatura Investigación Operativa cursada en el tercer curso del grado de Matemáticas.

siendo $\boldsymbol{w}$ una solución factible del problema dual 1.2. Observar que se verifican las CHC, ya que cuando $x_{ij} = 1$, $(i,j) \in \mathcal{P}$, por 1.4 se obtiene $c_{ij} - w_i + w_j$ y para todo $(i,j) \notin \mathcal{P}$, $x_{ij} = 0$. Así que aplicando el teorema 1.2 se tendrá que $\boldsymbol{x}$ es solución óptima, y por lo tanto $\mathcal{P}$ será el camino de ruta mínima desde 1 hasta N. □

Los algoritmos que se estudiarán en los próximos capítulos, resuelven el problema de ruta mínima, sin embargo, este problema, que se puede tratar como un problema de transbordo, también puede ser resuelto mediante el algoritmo de simplex. El problema con este algoritmo, es que tiene una complejidad computacional mucho mayor que los que se analizarán, por lo que en la práctica no es considerado óptimo para resolver este problema concreto.

En la figura 1.1 se muestra un sencillo ejemplo con 6 nodos y 9 arcos. Sobre los arcos se muestra los costos de conectar los nodos, y en verde se aprecia el camino dirigido de costo mínimo entre 1 y 6.

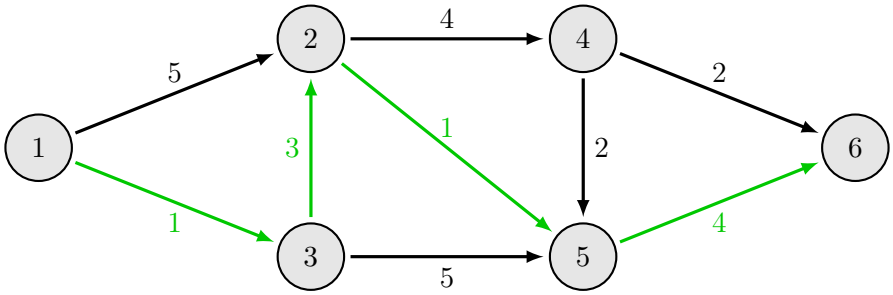

Figura 1.1: Ejemplo base.

1.2. Variantes del problema de ruta mínima

El problema de ruta mínima, tal y como se ha presentado, puede tener varias variantes. Las principales son las siguientes [3]:

- Ruta mínima entre dos únicos nodos.
- Ruta mínima desde un nodo a todos los demás.
- Ruta mínima desde todos los nodos a uno en concreto.
- Ruta mínima entre todos los pares de nodos del grafo.

De todas ellas, la que se considerará en este trabajo será la segunda, ya que los algoritmos que se estudiarán serán los de ruta mínima desde un nodo a todos los demás. Estos incluyen a los primeros, es decir, los algoritmos de rutas mínimas entre dos únicos nodos. Por otro lado, los algoritmos de ruta mínima desde todos los nodos a solamente uno de ellos se puede construir de forma simétrica a los que se plantearán. Finalmente, no se profundizará en los algoritmos que estudian las rutas entre todos los pares de nodos del grafo ya que exceden los límites del trabajo.

Para poder aplicar los algoritmos, se exigirá que el grafo del problema cumpla las siguientes condiciones.

- Es un grafo dirigido.
- Existe al menos un camino dirigido entre el nodo 1 y el resto (si no, solo calculará del nodo 1 a todos los alcanzables).

Además, para algunos algoritmos se requieren condiciones adicionales como las siguientes.

- El grafo no incluye costos negativos.
- Los costos c_{ij} asociados a cada arco (i,j) sean enteros.

1.3. Tipos de algoritmos

Para la resolución de los problemas de ruta mínima, se pueden hacer uso de una gran variedad de algoritmos. Todos ellos tienen en común su iteratividad, pero se puede diferenciar en dos grupos en función de qué método se utilice para actualizar las etiquetas de las distancias, además del criterio que se siga para eliminar nodos de la lista de candidatos. Antes de hacer la clasificación, se tendrá en cuenta una definición previa.

Definición 4. Se llama *lista de candidatos* al conjunto de nodos que pueden tenerse en consideración para mejorar la ruta mínima. Este conjunto, se define como V, siendo $V = \{1\}$ al inicio de las iteraciones. El algoritmo termina cuando $V = \emptyset$.

Una vez puesto el contexto, se puede pasar a los dos grupos de algoritmos mencionados previamente.

- **Algoritmos de asignación de etiquetas.** En estos algoritmos, los nodos eliminados de la lista de candidatos son los que tienen una etiqueta mínima, es decir, distancia mínima desde el nodo inicial. A su vez, todos los costos asociados a los arcos del grafo se pedirá que sean no negativas, evitando de este modo la presencia de ciclos negativos. Cada nodo solo pasará por V una vez, por tanto, dado un nodo que ya ha abandonado la lista de candidatos, su etiqueta será permanente hasta el final del algoritmo.
- **Algoritmos de corrección de etiquetas.** La elección de la eliminación del nodo i de la lista de candidatos es menos compleja que en el caso anterior, e intervienen menos cálculos. Cada algoritmo se diferencia en la forma en la que cada nodo entra o sale de esta al incorporarse o abandonar V respectivamente. Notar que los nodos podrán incorporarse a V más de una vez, tomando de esta forma las etiquetas de los nodos como temporales hasta la última iteración, donde todas las etiquetas tienen el valor óptimo. Estos algoritmos funcionan en presencia de ciclos negativos, siendo capaces de detectarlos e indicar que no existe ruta mínima.

Notar que los algoritmos de corrección de etiquetas se pueden usar para casos más generales que los de asignación de etiquetas, ya que permiten estudiar los grafos con costos negativos.

Capítulo 2

Algoritmo genérico

En este capítulo se presentará el método de resolución de problemas de ruta mínima denominado *algoritmo genérico*, el cual dependiendo de como se particularice, dará lugar a distintos algoritmos. Este algoritmo puede implementarse de diversas maneras, variando cada una de ellas, principalmente, en la selección de los nodos para ser eliminados de la lista de candidatos V. El desarrollo de este algoritmo y sus propiedades se basa en [4, 5, 6].

2.1. Algoritmo genérico

Como se ha mencionado anteriormente, se trata de un algoritmo que resuelve el problema de ruta mínima de un único nodo a todos los restantes en el grafo, y en consecuencia desde un nodo a otro específico. Se seguirá con la notación usada hasta ahora, y se tomará como nodo inicial 1, además de un vector de etiquetas $d = (d_1, d_2, \ldots, d_N)$. Cada d_i puede ser interpretado para todo i como la distancia del nodo 1 a i dada por un camino $\mathcal{P}_i$.

Para inicializar el algoritmo, se tiene en cuenta una lista de candidatos V, que en la primera iteración se definirá junto con d como

$$V = \{1\}, \quad d_1 = 0, \quad d_i = \infty \text{ para todo } i \neq 1,$$

es decir, la ruta actual hasta el nodo 1 tiene costo cero y para el resto aún no existe camino, por lo tanto sus costos serán $+\infty$. En cada iteración, se elimina un nodo i de la lista de candidatos. Para cada arco saliente de dicho nodo, $(i, j) \in \mathcal{A}$ con $j \neq 1$, se tiene que si se cumple la desigualdad $d_j > d_i + c_{ij}$ para dicho arco , entonces se fija una nueva distancia del nodo 1 a j que se define como se puede apreciar a continuación,

$$d_j := d_i + c_{ij},$$

satisfaciendo así las Condiciones de Holgura Complementaria dadas en la proposición 1.3, donde se afirma que $d_j \leq d_i + c_{ij}$, es decir, actualmente se podía llegar a j con un costo d_j, pero se ha encontrado que se puede llegar a i y posteriormente a j mediante el arco (i, j) con un costo nuevo $d_j = d_i + c_{ij}$ menos elevado. Este nodo j se añade a V, la lista de candidatos, si no lo estaba ya. El algoritmo continua hasta que las Condiciones de Holgura Complementaria se satisfacen para todos los arcos del grafo, es decir, terminará cuando la lista de candidatos esté vacía, es decir, cuando $V = \emptyset$.

Por todo el procedimiento explicado, el reemplazamiento del camino $\mathcal{P}_j$ por un camino más corto compuesto por $\mathcal{P}_i$ y el arco (i, j), puede verse como una operación de mejora de costos primales.

Puede apreciarse el pseudocódigo de este algoritmo más en detalle en el Algoritmo 1.

Algorithm 1: Algoritmo genérico.

$V \leftarrow \{1\}$
$d \leftarrow [0, \infty, \ldots, \infty]$
$M \leftarrow \min\{-1, (N-1) \min_{(i,j)\in\mathcal{A}} c_{ij}\}$

while $V \neq \emptyset$ **do**
  Se extrae un nodo i cualquiera de la lista
  for *todos los nodos* j *vecinos out de* i **do**
    if $d_j > d_i + c_{ij}$ **and** $j \neq 1$ **then**
      $d_j \leftarrow d_i + c_{ij}$
      if $d_j < M$ **then**
        STOP `/* Con M una cota para los ciclos negativos */`
      end
      if $d_j \notin V$ **then** se añade j a V
    end
  end
end

2.2. Propiedades del algoritmo genérico

En esta subsección se va a estudiar la corrección y convergencia del algoritmo genérico, y se realizará su aplicación manual en un sencillo ejemplo de ruta mínima.

La siguiente proposición justifica formalmente la corrección y convergencia del algoritmo genérico.

Proposición 2.1. *Se considera el problema de ruta mínima (P) y el algoritmo general aplicado a dicho problema. Se tendrán las siguientes propiedades:*

a) Dado $i \in \mathcal{N}$, $d_i < \infty$ *si y solo si* i *ha entrado al menos una vez a* V.

b) En cada iteración se puede afirmar:

  i) $d_1 = 0$.

  ii) Si $d_j < \infty$ *con* $j \neq 1$, *entonces* d_j *es la longitud de un camino que empieza en* 1, *nunca vuelve a* 1 *y termina en* j.

  iii) Si $i \notin V$, *entonces* $d_i = \infty$ *o* $d_j \leq d_i + c_{ij}$, *para todo* $(i,j) \in \mathcal{A}$.

c) Si el algoritmo termina, se tendrá entonces para todo j *con* $d_j < \infty$, d_j *es la distancia mínima del nodo* 1 *al nodo* j *dada por*

$$d_j = \begin{cases} \min\limits_{(i,j)\in\mathcal{A}} \{d_i + c_{ij}\} & \text{si } j \neq 1, \\ 0 & \text{si } j = 1. \end{cases} \tag{2.1}$$

d) Si el algoritmo no termina, existen caminos que empiezan en el nodo 1 *y terminan en* j, *que no vuelven al nodo inicial cuya longitud tiende a* $-\infty$.

Demostración.

a) Inicialmente, $d_1 = 0$ y $d_i = \infty$ para todo i distinto de 1 con $V = \{1\}$. En el transcurso del algoritmo, se fijan unas nuevas d_i monótonamente no crecientes y el nodo i queda añadido a su vez a V, por lo que queda así probada la propiedad.

b) i) Inicialmente se tiene $d_1 = 0$ y dado que por las normas del algoritmo, d_1 no podrá reetiquetarse, manteniendo así su valor constante.

ii) Se usará el método de inducción aplicado al número total de iteraciones del método. La propiedad se puede afirmar cierta para la primera iteración por definición del vector d, ya que $d_1 = 0$ y $d_i = \infty$ para todo i distinto de 1. Se supondrá cierta la propiedad para un cierto número de iteraciones en el que se obtiene como resultado la eliminación del nodo i de la lista de candidatos V.

Si $i \neq 1$ se tiene que $d_i < \infty$, que es cierto por (a). Aplicando la hipótesis de inducción se obtiene que d_i es la longitud de un camino $\mathcal{P}_i$ que empieza en 1, nunca vuelve a 1 y termina en i. Realizando la última iteración para un $j \neq 1$, el nuevo valor de d_j será el dado por la expresión $d_i + c_{ij}$ siendo en este caso d_j la longitud de un camino $\mathcal{P}_j$ que está compuesto por el camino $\mathcal{P}_i$ seguido del arco (i, j) que empieza en 1, termina en j y nunca vuelve a 1 por ser $j \neq 1$. Si $i = 1$, solo se puede dar en la primera iteración, por lo que se tendrá $d_j = c_{1j}$ para todos los nodos j que sean vecinos *out* de 1 y $d_j = \infty$ para el resto.

iii) Para cada i, se tiene por el apartado (a) que $d_i = \infty$ si i aún no ha entrado en la lista de candidatos. Si $d_i < \infty$, se satisface $d_j \leq d_i + c_{ij}$ para todo $(i, j) \in \mathcal{A}$. Hasta la siguiente entrada de i en V, d_i se mantiene constante, mientras que d_j para todo j con $(i, j) \in \mathcal{A}$ no puede aumentar, conservando así la condición inicial $d_j \leq d_i + c_{ij}$.

c) Se definen los conjuntos al finalizar el algoritmo

$$I = \{i | d_i < \infty\}, \quad \bar{I} = \{i | d_i = \infty\}.$$

Notar que $j \in \bar{I}$ equivale a que no exista camino $\mathcal{P}$ desde el nodo 1 hasta el j, ya que si $i \in I$, $i \notin V$ al finalizar el algoritmo y aplicando el apartado (b)(iii) se tendrá $j \in I$ para todo $(i, j) \in \mathcal{A}$. Consecuentemente, no existe camino $\mathcal{P}$ desde ningún nodo en I hasta un $j \in \bar{I}$, en concreto desde $1 \in I$. Y recíprocamente, si no hay trayectoria de 1 a j entonces por el apartado (b)(ii), no puede cumplirse $d_j < \infty$ y $j \in \bar{I}$.

Observar que al finalizar, para todo $j \in I, d_j$ es la distancia mínima, y cumple 2.1.

$$\text{Sea } i \in I, \qquad d_j \leq d_i + c_{ij}, \quad \text{ para todo } (i, j) \in \mathcal{A},$$

con d_i la longitud de un camino $\mathcal{P}_i$ que empieza en 1 y termina en i. Fijando un nodo $m \in I$ y aplicando la condición anterior a cada arco (i, j) de un camino cualquiera $\mathcal{P}$ desde 1 a m, se obtiene que su longitud es mayor o igual que $d_m - d_1 \overset{(b)(i)}{=} d_m$, por tanto la trayectoria que existe por (b)(ii) de costo d_m es la mínima, siendo P_m el camino más corto de 1 a m, por lo que en todos sus arcos (i, j) se tendrá $d_j = d_i + c_{ij}$, llegando así a $d_j = \min\limits_{(i,j)\in\mathcal{A}} \{d_i + c_{ij}\}$ para $j \in I$.

d) Si el algoritmo nunca termina, alguna etiqueta d_j decrece estrictamente infinitas iteraciones, generando distintos caminos $\mathcal{P}_j$ a su paso. Por propiedades de los grafos, estos caminos pueden descomponerse en un camino simple finito $\mathcal{P}_j$ del nodo 1 al j junto con una colección de ciclos negativos. Repitiendo este ciclo un número infinito de veces, se obtiene el resultado buscado, que el costo del camino tienda a $-\infty$.

□

Hasta ahora, no se ha impuesto ninguna condición adicional en el grafo representado por el problema, por lo que no se puede garantizar que este algoritmo termine. Para ello, desarrollando a partir de la proposición 2.1 (d), se tiene que el algoritmo terminará si y solo si no existe ningún camino que empiece en el nodo 1, que no vuelva a 1 y que contenga un ciclo negativo.

Todo esto se puede evitar detectando previamente la presencia de los ciclos negativos y deteniendo el algoritmo en este caso. Para ello se tendrá en cuenta que cuando para algún nodo k su respectivo d_k es menor que el límite inferior de la distancia de todos los caminos simples, se parará el algoritmo, es decir, si

$$d_k < (N-1) \min_{(i,j)\in\mathcal{A}, c_{ij}<0} c_{ij},$$

se tiene que en el camino $\mathcal{P}$ del nodo 1 al nodo k cuya longitud es igual a d_k, debe contener un ciclo negativo. Una vez comprobada esta propiedad, se garantiza la no existencia de ciclos negativos.

Si existe un camino desde el nodo inicial 1 a cada nodo j y a su vez, se asegura la no existencia de ciclos negativos, podrá afirmarse que el algoritmo termina y sus etiquetas finales serán finitas, dadas en la proposición 2.1 (c) ecuación 2.1. Esta ecuación se denomina *Ecuación de Bellman* y se trata de una formulación más general de las Condiciones de Holgura Complementaria 1.4 de la proposición 1.3, de donde se pueden obtener la distancia mínima de 1 a j como la suma de la distancia mínima de un nodo que precede a j y el arco que les conecta. Este mismo procedimiento se hace para calcular cualquier ruta mínima desde 1 hasta cualquier nodo j, volviendo hacia atrás hasta llegar al nodo inicial por los arcos correspondientes, obteniendo así un subgrafo conectado llamado *spanning tree*.

2.2.1. Inicialización avanzada

A pesar de que en este trabajo no se podrá analizar con más detenimiento, cabe destacar que este algoritmo no necesita cumplir inicialmente

$$V = \{1\}, \qquad d_1 = 0, \qquad d_i = \infty \text{ para todo } i \neq 1,$$

para poder funcionar de forma adecuada. Para que el algoritmo desempeñe su cometido de manera eficiente, bastará con que el conjunto de etiquetas $(d_1, \ldots, d_N)$ satisfaga las condiciones de la proposición 2.1 b. En particular, al algoritmo funcionará de forma adecuada si se asegura que V y d son inicializados con las siguientes condiciones más generales.

- Para cada nodo i, d_i tomará el valor ∞ o será la longitud de un cierto camino de 1 a i, con la excepción de $d_1 = 0$.
- La lista de candidatos V contiene todos los nodos i tal que

$$d_i + c_{ij} < d_j, \quad \text{con } (i,j) \in \mathcal{A}.$$

Esta técnica se usa en entornos de reoptimización, cuando se busca resolver problemas similares, o hacer una ligera modificación en un problema ya resuelto, ya sea en algún arco, o añadiendo o eliminando nodos. Podrán ahorrarse muchos costos computacionales usando las distancias de las rutas mínimas de un problema, como etiquetas iniciales de otro similar, ya que muchos no volverán a entrar la lista de candidatos V manteniendo así su distancia mínima.

Dado el grafo representado en la figura 1.1, aplicando el Algoritmo 1, se obtendrían en cada iteración los datos que se ven reflejados en la tabla 2.1, siendo los nodos marcados de color verde la lista de candidatos los que son añadidos a V, y las etiquetas en azul las que han cambiado en la iteración correspondiente. Este código de colores será el que se seguirá en las posteriores tablas del trabajo.

Para la primera iteración, se extrae el nodo 1 de la lista de candidatos y se examinan sus vecinos *out*, comprobando que la distancia a los nodos 2 y 3 son 5 y 1 respectivamente. Al ser menor las distancias nuevas que las que había en lista de etiquetas, se actualizan y se añaden

dichos nodos a V. Para la segunda iteración, se extrae un nodo de forma aleatoria de $V = \{2, 3\}$. Sea 3 el nodo que se extrae, se comprueban sus vecinos *out*, es decir, 2 y 5, las respectivas distancias serán 4 y 6, que se volverán a actualizar al ser menores, pero solo se añadirá el nodo 5 a la lista de candidatos ya que 2 ya se encontraba en ella. Para las siguientes iteraciones se hará de manera análoga.

Número de iteración	**Lista de candidatos (V)**	**Etiquetas de los nodos (d)**	**Nodo que sale de V**
1	{1}	(0,∞, ∞, ∞, ∞, ∞)	1
2	{2,3}	(0,5,1,∞, ∞, ∞)	3
3	{2,5}	(0,4,1,∞,6,∞)	5
4	{2,6}	(0,4,1,∞,6,10)	2
5	{6,4,5}	(0,4,1,8,5,10)	4
6	{6,5}	(0,4,1,8,5,10)	5
7	{6}	(0,4,1,8,5,9)	6
	$\emptyset$	(0,4,1,8,5,9)	

Tabla 2.1: Aplicación del algoritmo genérico.

Se puede apreciar que las distancias finales que calcula el algoritmo son las dadas cuando V es un conjunto vacío. En este caso, $d = (0, 4, 1, 8, 5, 9)$, que es el resultado que obtiene el algoritmo, es el que vector que representa la distancia mínima desde el nodo inicial a todos los demás del grafo. Se puede ver como a lo largo del algoritmo, las etiquetas van mejorando y puede volver a añadirse un nodo a V incluso después de haber sido eliminado de este.

Capítulo 3

Algoritmos de asignación de etiquetas

En este capítulo se estudiarán varias versiones de uno de los algoritmos más destacados para la resolución del problema de ruta mínima mediante la asignación de etiquetas, que asume que todos los costos asociados a los arcos del grafo $\mathcal{G}$ son no negativos, haciendo así que cada nodo solo entre la lista de candidatos una vez. Los desarrollos de este capítulo se basan principalmente en [5, 6].

3.1. Algoritmo de Dijkstra

Como se acaba de comentar, este algoritmo requiere obligatoriamente que $c_{ij} \geq 0$, es un caso especial del algoritmo genérico, donde los nodos son eliminados de la lista de candidatos V siguiendo un patrón determinado.

La idea que subyace en este algoritmo es que dado que todos los costos son mayores o iguales que cero, se van explorando todas las rutas más cortas que se inician desde el nodo 1 y llevan a todos los nodos del grafo, deteniéndose el algoritmo cuando se haya encontrado el camino más corto a todos sus nodos.

Parte de las mismas condiciones iniciales para la primera iteración que el algoritmo genérico, es decir,

$$V = \{1\}, \qquad d_1 = 0, \qquad d_i = \infty \text{ para todo } i \neq 1,$$

junto con la restricción del vector de costos, que tendrá que cumplir $c_{ij} \geq 0$. En cada iteración se elimina un nodo i de la lista de candidatos que sigue el siguiente criterio:

$$d_i = \min_{j \in V} \{d_j\}.$$

Para dicho nodo i, se toman sus arcos salientes $(i, j) \in \mathcal{A}$ con $j \neq 1$ y se comprueba si $d_j > d_i + c_{ij}$. En caso de verificarse, se redefinirá d_j como

$$d_j := d_i + c_{ij},$$

y posteriormente se añadirá dicho nodo j a la lista de candidatos V en caso de que aún no pertenezca. Al igual que el algoritmo anterior, seguirá haciendo iteraciones hasta que V sea un conjunto vacío.

El pseudocódigo del método es el que se desarrolla en el Algoritmo 2.

Para poder hacer un análisis en profundidad de este método y dar unas propiedades más específicas, será necesario previamente la definición del conjunto de nodos que ya han estando en la lista de candidatos pero no lo están actualmente.

$$W := \{i | d_i < \infty, i \notin V\} \tag{3.1}$$

Este conjunto, permite definir una serie de propiedades a cerca del algoritmo que se pueden ver en la siguiente proposición.

Algorithm 2: Algoritmo Dijkstra.

$V \leftarrow \{1\}$
$d \leftarrow [0, \infty, \ldots, \infty]$
while $V \neq \emptyset$ **do**
  for $j \in V$ **do**
    Se toma el mínimo de los d_j
    Se extrae de V dicho j
  end
  for *todos los nodos* j *vecinos out de* i **do**
    if $d_j > d_i + c_{ij}$ **and** $j \neq 1$ **then**
      $d_j \leftarrow d_i + c_{ij}$
    end
    if $d_j \notin V$ **then** Se añade j a V
  end
end

Proposición 3.1. *Considerando el problema de ruta mínima (P) descrito en 1.1 y a su vez asumiendo* $c_{ij} \geq 0$ *y la existencia de al menos un camino* $\mathcal{P}$ *desde el nodo* 1 *al resto de los nodos, se tienen las siguientes propiedades*[1].

a) Para cada iteración del algoritmo, dado el conjunto W *definido en 3.1, se puede afirmar:*

  i) Ningún nodo $i \in W$ *vuelve a entrar a la lista de candidatos.*

  ii) Al final de cada iteración, se tiene

$$d_i \leq d_j, \qquad \text{para todo } i \in W,\ j \notin W.$$

  iii) Para cada nodo i*, sea* $\mathcal{P}$ *un camino que empieza en* 1*, termina en* i *y tiene todos sus nodos en* W *al final de la iteración. Se tendrá entonces que la etiqueta* d_i *al final de la iteración es igual a la longitud del camino más corto de los dados. En particular, si* $d_i = \infty$*, dicho camino no existirá.*

b) Todos los nodos serán eliminados de la lista de candidatos solamente una vez, teniendo en cuenta que i *será eliminado antes que* j *si* $d_i < d_j$*.*

W puede interpretarse como un conjunto de nodos con etiquetas permanentes, ya que dado un nodo i que entra en W, $i \in W$ hasta el final del algoritmo, manteniendo así su etiqueta constante al no volver a entrar a V, es decir, el algoritmo finaliza cuando se han hecho permanentes todas las etiquetas de los nodos.

Nota 1. Se puede comprobar juntando la proposición 3.1 (a)(ii) con $c_{ij} \geq 0$, que en efecto para cada nodo i que es eliminado de V,

$$d_j \leq d_i + c_{ij}, \qquad \text{para todo } j \in W \text{ tal que } (i,j) \in \mathcal{A},$$

cumple las Condiciones de Holgura Complementaria.

Se vuelve a tomar el grafo representado en la figura 1.1, sobre el que se aplicará el algoritmo de Dijkstra, y se podrá ver representado en la tabla 3.1 los pasos que sigue, permitiendo así una comparación con el algoritmo anterior.

Las distancias finales que calcula el algoritmo son las dadas cuando la lista de candidatos se encuentra vacía, que vuelven a ser $d = (0, 4, 1, 8, 5, 9)$, representando así la ruta mínima desde

[1]Este algoritmo y sus propiedades fueron estudiadas en la asignatura Grafos y Combinatoria, correspondiente al primer curso del grado de Matemáticas, por lo que la demostración será omitida.

Número de iteración	**Lista de candidatos (V)**	**Etiquetas de los nodos (d)**	**Nodo que sale de V**
1	{1}	(0,∞,∞,∞,∞,∞)	1
2	{2,3}	(0,5,1,∞,∞,∞)	3
3	{2,5}	(0,4,1,∞,6,∞)	2
4	{5,4}	(0,4,1,8,5,∞)	5
5	{4,6}	(0,4,1,8,5,9)	4
6	{6}	(0,4,1,8,5,9)	6
	$\emptyset$	(0,4,1,8,5,9)	

Tabla 3.1: Aplicación del algoritmo de Dijkstra.

el nodo inicial 1 hasta el resto de los nodos. En este caso, las nodos que abandonan la lista de candidatos no vuelven a ser añadidos a esta, quedándose así con su etiqueta fija.

El número de iteraciones del algoritmo será igual al número de nodos del grafo (N). Cada iteración se basará en dos operaciones. La primera será la elección del nodo a extraer de la lista de candidatos, que a lo sumo tendrá un costo computacional de $O(N)$, que al repetirlo en cada iteración acabará siendo un total de $O(N^2)$ operaciones. La otra operación que se tiene en cuenta es el ajuste de las etiquetas, que en cada iteración el número de operaciones difiere, ya que el algoritmo tiene que comprobar todos los arcos salientes. Sin embargo, teniendo en cuenta todas las iteraciones, el algoritmo comprobará todos los arcos del grafo, por lo que este computo se dirá que se lleva a cabo un total de m veces, siendo el número de operaciones $O(m)$, que se despreciará por ser menor que $O(N^2)$, concluyendo que la complejidad del algoritmo de Dijkstra para resolver el problema de ruta mínima es de $O(N^2)$.

En los siguientes apartados se llevan a cabo unas modificaciones en el algoritmo de Dijkstra que se realizan con objeto de mejorar la complejidad computacional del algoritmo. Estas modificaciones actuarán en la búsqueda de la etiqueta mínima de los nodos en la lista de candidatos, la complejidad, $O(N^2)$, se puede reducir usando unas estructuras de datos más adecuada. Sin embargo, $O(m)$, que es el número de operaciones para llevar a cabo el ajuste de etiquetas, no podrá ser reducido. Estas modificaciones y las alteraciones que producen cada uno de ellos al algoritmo, se analizarán de una manera mas exhaustiva a continuación.

3.2. Algoritmo de Dijkstra con cola de prioridad

Este algoritmo, también conocido como algoritmo de *Heap*, se trata de una modificación del algoritmo de Dijkstra en el que se obtendrá como resultado un coste computacional menor haciendo uso de pilas, que son comúnmente usadas para implementar colas de prioridad, estas se utilizan para almacenar eficientemente la lista de candidatos por su valor de d_j.

En este caso, se usará una pila binaria basándose en las etiquetas de los nodos y su pertenencia a la lista de candidatos V. El nodo situado en lo más alto de la pila, será el nodo en V cuya etiqueta tiene el menor valor, se le denomina la raíz. Notar también que las etiquetas de todos los nodos pertenecientes a la lista de candidatos, nunca serán superiores que las etiquetas de todos sus nodos descendientes que están a su vez en V. Por otra parte, los nodos que no pertenecen a la lista de candidatos estarán en la pila, aunque puede que no tengan ningún descendiente perteneciente a V.

En cada iteración el nodo en la lista de candidatos con menor etiqueta será distinto, ya que el anterior habrá sido extraído de V, fijando así su etiqueta permanentemente. Si la etiqueta de algún nodo decrece en esta iteración, entonces este deberá mover su posición en la pila a una

más cercana a la raíz. Asimismo, si entra en la lista de candidatos un nodo nuevo, tendrá que ser colocado en la pila en la posición adecuada.

En este caso no se incorpora el pseudocódigo, únicamente se indica que el proceso de incorporación y borrado de nodos en la lista de candidatos se realiza con una estructura de pila. En la implementación se ha utilizado la librería `heapq` de *Python*. Todo lo necesario viene definido dentro de la librería, por lo que con tan solo añadir unas listas de tuplas y modificar las funciones asociadas a estos objetos a las correspondientes de la nueva librería, basta para tener el algoritmo de Dijsktra con colas de prioridad.

Esta modificación en la búsqueda de la etiqueta mínima de los nodos, hace que disminuya la complejidad computacional. Se tendrá en cuenta que cada operación de eliminación, reubicación o incorporación de nodos a la pila supone una complejidad de $O(logN)$. Por lo que, al eliminarse un nodo en cada iteración, y ser N el número total de nodos eliminados, supondrá una complejidad de $O(NlogN)$. Siendo el número total de nodos reubicados a lo sumo m, se tendrá un coste de $O(mlogN)$. Si se tienen en cuenta el número de operaciones que se llevan a cabo para el ajuste de la etiqueta, $O(m)$, se llega a que la complejidad del algoritmo será $O(mlogN)$.

3.3. Algoritmo de Dial

Es otra modificación del algoritmo de Dijkstra, que disminuye su coste computacional haciendo uso de *buckets* en la lista de candidatos, creando así una división de la lista.

La idea del algoritmo es guardar para cada posible valor de las etiquetas una lista de los nodos con ese valor. Será necesario que todos los arcos tengan un costo entero no negativo. Notar que, al ser todos los costos positivos, una etiqueta finita representa la longitud de un camino sin ciclos, por lo que las posibles etiquetas estarán en el rango de 0 a $(N-1)C$, siendo C el valor máximo de todos costos asociados a los arcos. Habrá así $(N-1)C+1$ posibles valores de las etiquetas y a su vez *buckets* que serán examinados en orden ascendente hasta encontrar uno no vacío.

Los *buckets* (B_k) pueden pensarse como contenedores en los que están las etiquetas en un rango de $[0,(N-1)C]$. Cada B_k contiene nodos cuya etiqueta tiene valor k.

Inicialmente, se pone 1 en B_0 y el resto de *buckets* estarán vacíos. En la primera iteración, para todo nodo j con $(1,j) \in \mathcal{A}$ se añade al *bucket* $B_{c_{1j}}$ con etiqueta c_{1j} y a la lista de candidatos V. Una vez comprobado el B_0, se examinará B_1. Si está vacío se continua revisando B_2, y posteriores en orden ascendente, si no, se repite el proceso tomando y eliminando de la lista de candidatos un nodo de etiqueta 1 y moviendo los nodos cuya etiqueta haya cambiado a los correspondientes *buckets* de nodos con etiquetas menores. Tener en cuenta que cada iteración se empieza desde el *bucket* del que fue extraído el anterior nodo. Como resultado, una vez que se vacía un B_k no se vuelve a examinar, ya que el nodo eliminado, al tratarse de un algoritmo Dijkstra tiene la menor etiqueta del resto de nodos en V, y en las siguientes iteraciones no se fijaran valores más pequeños al tratarse de costos no negativos.

Nº iteración	**V**	**Etiquetas de los nodos (d)**	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	**Nodo que sale de V**
1	{1}	(0,∞,∞,∞,∞,∞)	1	-	-	-	-	-	-	-	-	-	1
2	{2,3}	(0,5,1,∞,∞,∞)	-	3	-	-	-	2	-	-	-	-	3
3	{2,5}	(0,4,1,∞,6,∞)	-	-	-	-	2	-	5	-	-	-	2
4	{5,4}	(0,4,1,8,5,∞)	-	-	-	-	-	5	-	-	4	-	5
5	{4,6}	(0,4,1,8,5,9)	-	-	-	-	-	-	-	-	4	6	4
6	{6}	(0,4,1,8,5,9)	-	-	-	-	-	-	-	-	-	6	6
	∅	(0,4,1,8,5,9)											

Tabla 3.2: Aplicación del algoritmo Dial.

En la tabla 3.2 se puede ver aplicando el algoritmo 3 sobre el grafo de la figura 1.1.

Se puede apreciar que da el mismo resultado en las mismas iteraciones que en al tabla 3.1, pero con la diferencia de que las etiquetas de este algoritmo están organizadas por grupos de su misma etiqueta. Sin embargo, este pequeño cambio hace que el número de operaciones sea menor que el original, ya que se busca el menor B_k no vacío, en lugar de analizar toda la lista de candidatos en busca del mínimo. En cada iteración, habrá que comprobar si el *bucket* se encuentra vacío y si es necesario añadir o quitar nodos, suponiendo cada operación un costo de $O(1)$. El costo de buscar el mínimo para cada iteración, de forma global cuesta $O(NC)$ ya que hay $(N-1)C+1$ *buckets*. Además, el número de operaciones para el ajuste de etiquetas tras actualizar los d_j, se ve reflejado en la colocación de cada nodo en su *bucket* correspondiente y la reposición de estos en cada iteración se tendrán $O(m)$ operaciones, que junto con las anteriores, se llega a un coste computacional total de $O(m+NC)$, en donde, si C no es muy grande, comparado con N o $logN$, mejora la complejidad original de los algortimos de Dijkstra y Dijkstra con colas de prioridad.

Algorithm 3: Algoritmo Dial.

```
V ← {1}
d ← [0, ∞, ..., ∞]
B ← [[1][ ][ ] ... [ ]]
i ← 0
while V ≠ ∅ do
    while B[i] = ∅ do
        Se pasa al siguiente bucket
    end
    Se extrae un nodo j de V
    for todos los elementos j vecinos out de i do
        if d_j > d_i + c_ij and j ≠ 1 then
            if d_j < ∞ then
                Extraer j de B[d_j]
            end
            d_j ← d_i + c_ij
            Se añade j a B[d_j]
        end
        if d_j ∉ V then se añade j a V
    end
end
```

Capítulo 4

Algoritmos de corrección de etiquetas

Los algoritmos de corrección de etiquetas usan métodos menos sofisticados para realizar la eliminación del nodo i de la lista de candidatos, que irá unido a un menor costo computacional. Sin embargo, a diferencia de los algoritmos de asignación de etiquetas, un nodo puede entrar varias veces en la lista de candidatos. La mayor diferencia que se encuentra en este tipo de algoritmos es la forma de gestionar la lista de candidatos para añadir y extraer nodos. Para el desarrollo de los algoritmos a continuación se ha hecho uso de [5, 6].

4.1. Algoritmo de Bellman-Ford

Se trata de un algoritmo que resuelve la ruta mínima desde un nodo inicial 1 a los demás nodos. Es notable destacar que este algoritmo sí que permite la existencia de costos negativos.

Este método calcula el costo del camino óptimo desde el nodo inicial 1 hasta j para todo $j \in \mathcal{N}$, usando como máximo tantos arcos como número de iteración en la que se encuentra el algoritmo. Será en la última iteración cuando se fijan los valores finales de la etiquetas, hasta entonces son tratados como temporales.

Las condiciones iniciales del algoritmo serán las siguientes,

$$V = \{1\}, \qquad d_1 = 0, \qquad d_i = \infty \text{ para todo } i \neq 1,$$

al igual que en los algoritmos previos. La lista de candidatos sigue un orden *FIFO (First In First Out)*, por lo que el primer nodo en entrar será el primero en abandonarla. En cada iteración, se eliminarán de la lista de candidatos todos los nodos añadidos en el paso anterior, actualizando así V en cada iteración, esto puede verse como si el algoritmo trabajara en ciclos de iteraciones. Para cada nodo i que se extrae de V, se examinan todos los arcos $(i, j) \in \mathcal{A}$, comprobando si $d_j > d_i + c_{ij}$, que en caso de verificarse, se fijará una nueva etiqueta para el nodo j,

$$d_j := d_i + c_{ij},$$

y se añadirá el nodo j al final de la lista de candidatos correspondiente con el nuevo ciclo. Se harán N iteraciones, donde la última es una comprobación de la existencia de ciclos en el grafo, dado que si las etiquetas se actualizan en esta última iteración, habrá un ciclo negativo, de lo contrario, las etiquetas resultantes serán la solución buscada.

En el Algoritmo 4 se muestra el pseudocódigo de este método.

Con el fin de presentar el funcionamiento del algoritmo, será necesario una definición previa a la proposición, la cual destaca una propiedad que se distingue sobre el resto.

Definición 5. Se denominará como d_i^k a la distancia más corta desde el nodo 1 hasta el nodo i usando caminos con k arcos o menos. En particular, $d_i^k = \infty$ si no existe un camino de 1 a i con k arcos o menos, siendo $d_1^0 = 0$ y $d_i^0 = \infty$ para todo $i \neq 1$.

Algorithm 4: Algoritmo Bellman-Ford.

```
V ← {1}
d ← [0, ∞, ..., ∞]
for i in range (N) do
    it ← 0
    lV ← len(V)
    while it<lV do
        it ← it + 1
        Se extrae el primer elemento de V
        for todos los nodos j vecinos out de i do
            if d_j > d_i + c_ij then
                if i == N − 1 then  Existe un ciclo de costo negativo
                d_j ← d_i + c_ij
            end
            if d_j ∉ V then  Se añade j a V        /* Si j no ha sido añadido */
                                                   /* a V en el ciclo actual */
        end
    end
end
```

Proposición 4.1. *Propiedad de Bellman-Ford Para cada nodo i y $k \geq 1$, $k \in \mathbb{N}$, se tendrá que al final del $k-$ésimo ciclo de iteraciones del método de Bellman-Ford, $d_i \leq d_i^k$.*

Demostración. Notar que, dada la definición 5, d_j^{k+1} es la distancia más corta desde el nodo 1 hasta el nodo i usando caminos con $k+1$ arcos o menos. Es decir, será o la longitud de un camino desde el nodo 1 al j con k arcos o menos, es decir d_j^k, o en su defecto, la longitud de un camino desde el nodo 1 a un predecesor de j, el nodo i y luego llega a j usando el arco $(i,j) \in \mathcal{A}$. Esto es,

$$d_j^{k+1} := \text{mín}\left\{ d_j^k, \min_{(i,j)\in\mathcal{A}} \{d_i^k + c_{ij}\} \right\}, \qquad \text{para todo } j,k \geq 1. \tag{4.1}$$

Se probará la propiedad mediante el método de inducción. Una vez finalizado el primer ciclo de iteraciones, se tendrá para todo nodo i,

$$d_i = \begin{cases} 0 & \text{si } i = 1, \\ c_{1i} & \text{si } i \neq 1 \text{ y } (1,i) \in \mathcal{A}, \\ \infty & \text{si } i \neq 1 \text{ y } (1,i) \notin \mathcal{A}, \end{cases} \qquad d_i^1 = \begin{cases} c_{1i} & \text{si } (1,i) \in \mathcal{A}, \\ \infty & \text{si } (1,i) \notin \mathcal{A}, \end{cases}$$

de donde se deduce $d_i \leq d_i^1$ para todo $i \in \mathcal{N}$. Se supondrá cierto $d_i \leq d_i^k$ para todo nodo i. Se tomará d_i y V como las etiquetas de los nodos y la lista de candidatos en el $k-$ésimo ciclo respectivamente, mientras que $\bar{d}_i$ indicará las etiquetas en los nodos al final del ciclo $k+1$. Se busca probar así que $\bar{d}_i \leq d_i^{k+1}$. Por la proposición 2.1 (b)(iii) se sabe

$$d_j \leq \ d_i + c_{ij}, \qquad \text{para todo } (i,j) \in \mathcal{A}, i \notin V. \tag{4.2}$$

Aplicando que $\bar{d}_j \leq d_j$, se transforma 4.2 en

$$\bar{d}_j \leq \ d_i + c_{ij}, \qquad \text{para todo } (i,j) \in \mathcal{A}, i \notin V. \tag{4.3}$$

Por otro lado, se tiene 4.4 ya que cuando el nodo i es eliminado de la lista de candidatos, la etiqueta actual denominada $\tilde{d}_i$ satisface $\tilde{d}_i \leq d_i$, y la etiqueta de j es fijada a $\tilde{d}_i + c_{ij}$ si excede $\tilde{d}_i + c_{ij}$.

$$\bar{d}_j \leq d_i + c_{ij}, \qquad \text{para todo } (i,j) \in \mathcal{A}, i \in V. \tag{4.4}$$

Combinando así las ecuaciones 4.3 y 4.4,

$$\bar{d}_j \leq \min_{(i,j)\in\mathcal{A}} \{d_i + c_{ij}\} \overset{H.I.}{\leq} \min_{(i,j)\in\mathcal{A}} \{d_i^k + c_{ij}\}, \qquad \text{para todo } j.$$

A su vez, se tiene $\bar{d}_j \leq d_j \overset{H.I.}{\leq} d_j^k$, de donde se obtiene el paso final

$$\bar{d}_j \leq \min\left\{d_j^k, \min_{(i,j)\in\mathcal{A}} \{d_i^k + c_{ij}\}\right\} \overset{4.1}{=} d_j^{k+1}.$$

□

Notar que el algoritmo no terminará si y solo si existe un camino que empieza en el nodo 1 y contiene ciclos negativos.

Si todos los costos del grafo son no negativos, se puede asegurar que el algoritmo calculará el camino dirigido más corto para cada nodo en un máximo de $N-1$ iteraciones, dado que son caminos simples, pudiendo contener a lo sumo los N nodos del grafo. Por la propiedad de Bellman-Ford 4.1 se puede concluir que si el algoritmo termina después de $N-1$ iteraciones, existe un ciclo negativo en el grafo.

Número de iteración	**Lista de candidatos (V)**	**Etiquetas de los nodos(d)**
1	{1}	(0,5,1,∞, ∞, ∞)
2	{2,3}	(0,4,1,9,6,∞)
3	{4,5,2}	(0,4,1,8,5,10)
4	{6,4,5}	(0,4,1,8,5,9)
5	{6}	(0,4,1,8,5,9)
6	$\emptyset$	(0,4,1,8,5,9)

Tabla 4.1: Aplicación del algoritmo de Bellman-Ford.

En la tabla 4.1 se puede ver la aplicación del algoritmo 4.1 sobre el grafo representado en 1.1. Las etiquetas que aparecen en cada iteración en dicha tabla son las ya actualizadas al finalizar cada paso. En este ejemplo se puede ver que en efecto los nodos entran varias veces la lista de candidatos.

Por otra parte, para calcular el coste computacional, se tendrá en cuenta que en el peor de los casos en cada ciclo de iteración el algoritmo tendrá una complejidad $O(m)$ de actualizar etiquetas y de $O(N)$ de tomar los nodos de la lista, siendo el ciclo de iteración una complejidad $O(m)$. Teniendo en cuanta el número total de ciclos, que a lo sumo serán N, se obtiene un coste computacional de $O(mN)$.

4.2. Algoritmo D'Esopo-Pape

Se trata también de una modificación del algoritmo genérico, por lo que las condiciones iniciales serán las mismas, no obstante la lista de candidatos seguirá un orden específico.

El nodo extraído será el primero de la lista, sin embargo para los nodos que se añaden a V se distinguirá en dos casos, si ese nodo nunca ha estado en la lista de candidatos, se añadirá al final esta, por el contrario, si ya ha estado en V, el nodo será añadido al principio de este mismo. Se

hace de esta manera ya que cuando se elimina un nodo i de V, su etiqueta afectará a los nodos j tal que $(i,j) \in \mathcal{A}$ pudiendo verse modificadas así sus correspondientes etiquetas. Si se vuelve a renovar la etiqueta del nodo i, es posible que también lo hagan la de los j dados, por lo que serán añadidos al principio de la lista, para poder así actualizar las etiquetas lo más rápido posible.

Algorithm 5: Algoritmo D'Esopo-Pape.

$V \leftarrow \{1\}$
$d \leftarrow [0, \infty, \ldots, \infty]$
$M \leftarrow \min\{-1, (N-1) \min_{(i,j)\in\mathcal{A}} c_{ij}\}$
while $V \neq \emptyset$ **do**
  Se extrae el primer elemento de V
  for *todos los nodos* j *vecinos out de* i **do**
    if $d_j > d_i + c_{ij}$ **and** $j \neq 1$ **then**
      if $d_j \notin V$ **then**
        if $d_j < \infty$ **then** Se añade j al comienzo de V
        else Se añade j al final de V
      end
      $d_j \leftarrow d_i + c_{ij}$
      if $d_j < M$ **then** STOP
    end
  end
end

La resolución del problema habitual se muestra en la tabla 4.2. En esta tabla se ve en la lista de candidatos los nodos que entran marcados en color, distinguiendo entre el verde para los que entran por primera vez y se colocan al final de la fila y el rojo para los nodos que ya habían entrado en V, colocándose así en cabeza. Al ser un algoritmo de corrección de etiquetas, se puede apreciar como hay nodos que entran la lista de candidatos varias veces.

Número de iteración	**Lista de candidatos (V)**	**Etiquetas de los nodos (d)**	**Nodo que sale de V**
1	{1}	(0,∞, ∞, ∞, ∞, ∞)	1
2	{2,3}	(0,5,1,∞, ∞, ∞)	2
3	{3,4,5}	(0,5,1,9,6,∞)	3
4	{2,4,5}	(0,4,1,9,6,∞)	2
5	{4,5}	(0,4,1,8,5,∞)	4
6	{5,6}	(0,4,1,8,5,10)	5
7	{6}	(0,4,1,8,5,10)	6
	∅	(0,4,1,8,5,9)	

Tabla 4.2: Aplicación del algoritmo D'Esopo-Pape.

Este algoritmo tiene una complejidad exponencial debido que el número de entradas de algunos nodos a la lista de candidatos puede llegar a ser no polinomial. En [7] y [8] se proponen problemas de ruta mínima donde este hecho se pone en manifiesto. Sin embargo, existen variantes de este algoritmo con complejidades polinómicas, estas pueden consultarse en [9] y [10].

4.3. Algoritmo *Small Label First* (SLF)

Este algoritmo intenta colocar los nodos con etiquetas más pequeñas al comienzo de la lista de candidatos. Cuando todos los costos asociados a los arcos son no negativos, suele reducir el número de veces que entra un nodo a la lista de candidatos [11].

Se trata de una modificación del algoritmo genérico, siendo así las condiciones iniciales las mismas. En cada iteración se extrae de la lista de candidatos el primer nodo, mientras que el nodo j que se añade a V, será comparado con el nodo inicial de la lista de candidatos mediante sus respectivas etiquetas. Sea i el primer nodo de V, j es añadido al inicio de V si $d_j \leq d_i$, en su defecto, j es añadido al final de la lista de candidatos.

Notar que cuanto más pequeña fuera la etiqueta del nodo j en la anterior extracción de la lista de candidatos, será menos probable que se actualice d_j. En particular, si $c_{ij} \geq 0$ y además,

$$d_j \leq \min_{i \in V} d_i,$$

ningún $i \in V$ tal que $(i, j) \in \mathcal{A}$ cumplirá $d_i + c_{ij} < d_j$.

Este método simula la política del algoritmo de Dijkstra de selección del nodo con etiqueta mínima pero con menos operaciones, además de ser aptos para costos negativos.

Se trata de un algoritmo cuya complejidad computacional es no polinomial, aunque se puede construir una versión del algoritmo que tendrá complejidad polinomial, $O(Nm^2)$ [12].

En la tabla 4.2 se puede ver la aplicación del Algoritmo 6 sobre el problema habitual. Los nodos de color rojo son los que se añaden al principio de la lista de candidatos por ser su etiqueta menor que la de su primer nodo, mientras que los de color verde son los que su etiqueta es menor.

Algorithm 6: Algoritmo SLF.

```
V ← {1}
d ← [0, ∞, ..., ∞]
M ← mín{−1, (N − 1) mín_{(i,j)∈A} c_ij}
while V ≠ ∅ do
    Se extrae el primer elemento de V
    for todos los nodos j vecinos out de i do
        if d_j > d_i + c_ij and j ≠ 1 then
            d_j ← d_i + c_ij
            if d_j < M then
                STOP
            end
            if len(V) == 0 then
                Se añade j a V
            else if d_j ∉ V then
                if d_j ≤ d_i then Se añade j al comienzo de V
                else Se añade j al final de V
            end
        end
    end
end
```

Número de iteración	**Lista de candidatos (V)**	**Etiquetas de los nodos (d)**	**Nodo que sale de V**
1	{1}	(0,∞, ∞, ∞, ∞, ∞)	1
2	{3,2}	(0,5,1,∞, ∞, ∞)	3
3	{2,5}	(0,4,1,∞,6,∞)	2
4	{5,4}	(0,4,1,8,5,∞)	5
5	{4,6}	(0,4,1,8,5,9)	4
6	{6}	(0,4,1,8,5,9)	6
	$\emptyset$	(0,4,1,8,5,9)	

Tabla 4.3: Aplicación del algoritmo SLF.

Capítulo 5

Estudio computacional

En este capítulo se lleva a cabo la comparación del comportamiento de los algoritmos estudiados a lo largo del trabajo mediante unos problemas test, creados a través de un generador de redes de problema de ruta mínima[1] proporcionado por el director del Trabajo de Fin de Grado. Los algoritmos se han implementado utilizando el lenguaje de programación *Python* [13].

Este generador crea redes en forma de rejilla, que contendrán $l \times k + 2$ nodos, correspondiendo a las filas l y columnas k, respectivamente. A dicha red se le añaden dos nodos que se unen con la primera y última columna, siendo así el nodo inicial y final. Los arcos tienen una probabilidad mayor de avanzar hacia delante en la red, aunque tienen la posibilidad de retroceder en las columnas o quedarse en la misma que ya estaban. El número total de arcos se establece en proporción a la densidad de la red, es decir, se proporciona una densidad α y se generan $\alpha \cdot n(n-1)$ arcos, una proporción α sobre el número máximo de aristas posibles, $n(n-1)$, que son distribuidos de forma uniforme.

Por ejemplo, para una red con 30 nodos, el número máximo de arcos es 870, y tomando una densidad del 5 % se tendrá una red con $43.5 \approx 44$ arcos, que supondrá que de cada nodo saldrán 1 o 2 arcos.

En el estudio, se va a tener en cuenta el número de nodos, densidad y forma de la red. Para empezar se diferenciará en 3 cantidades de nodos diferentes, que serán $100, 500$ y 1000. Por otro lado, se aplicarán tres densidades diferentes, $5\,\%, 10\,\%$ y $25\,\%$, y por último se distinguirá 7 formas de redes. Con el objetivo de simplificar la notación en la representación de la forma, se tomará $a \times b$ siendo la proporción entre l y k respectivamente, así las formas de las redes tomadas serán $1 \times 6, 1 \times 4, 1 \times 2, 1 \times 1, 2 \times 1, 4 \times 1, 6 \times 1$. Para cada tipo de red, se generan 30 problemas con costos positivos y se resuelven con cada algoritmo estudiado, guardando el tiempo de ejecución en milisegundos para su posterior estudio con *R-Commander*, donde se generarán las gráficas y tablas de las figuras 5.1 y 5.2 y del Anexo B.

En la figura 5.1 se representa el tiempo medio de ejecución de cada algoritmo, diferenciando entre el número de nodos, densidad, y formato. A continuación se comentará el comportamiento de los algoritmos en las distintas situaciones dadas.

Según el número de nodos, es decir, comparando la evolución de las gráficas de la figura 5.1 a través de las diferentes filas, se aprecia que el algoritmo general es el peor, sobretodo para densidades mayores. La variación de esta variable provoca un aumento en el tiempo de ejecución de los algoritmos significativo. El algoritmo Dial, con pocos nodos, obtiene tiempos mayores que el resto de algoritmos, pero a medida que va aumentando el número de nodos, sus resultados mejoran, convirtiéndose en un algoritmo muy competente. Esto podría reflejar el término lineal en el número de nodos de su complejidad.

Por el contrario, el algoritmo Heap, se comporta adecuadamente con un número de nodos inferior, pero a medida que aumenta, empeora drásticamente, lo que se puede deber a que la implantación del paquete `heapq` no esté muy optimizada. El resto de algoritmos se comportan

[1] En el Anexo B se explica brevemente el funcionamiento del generador

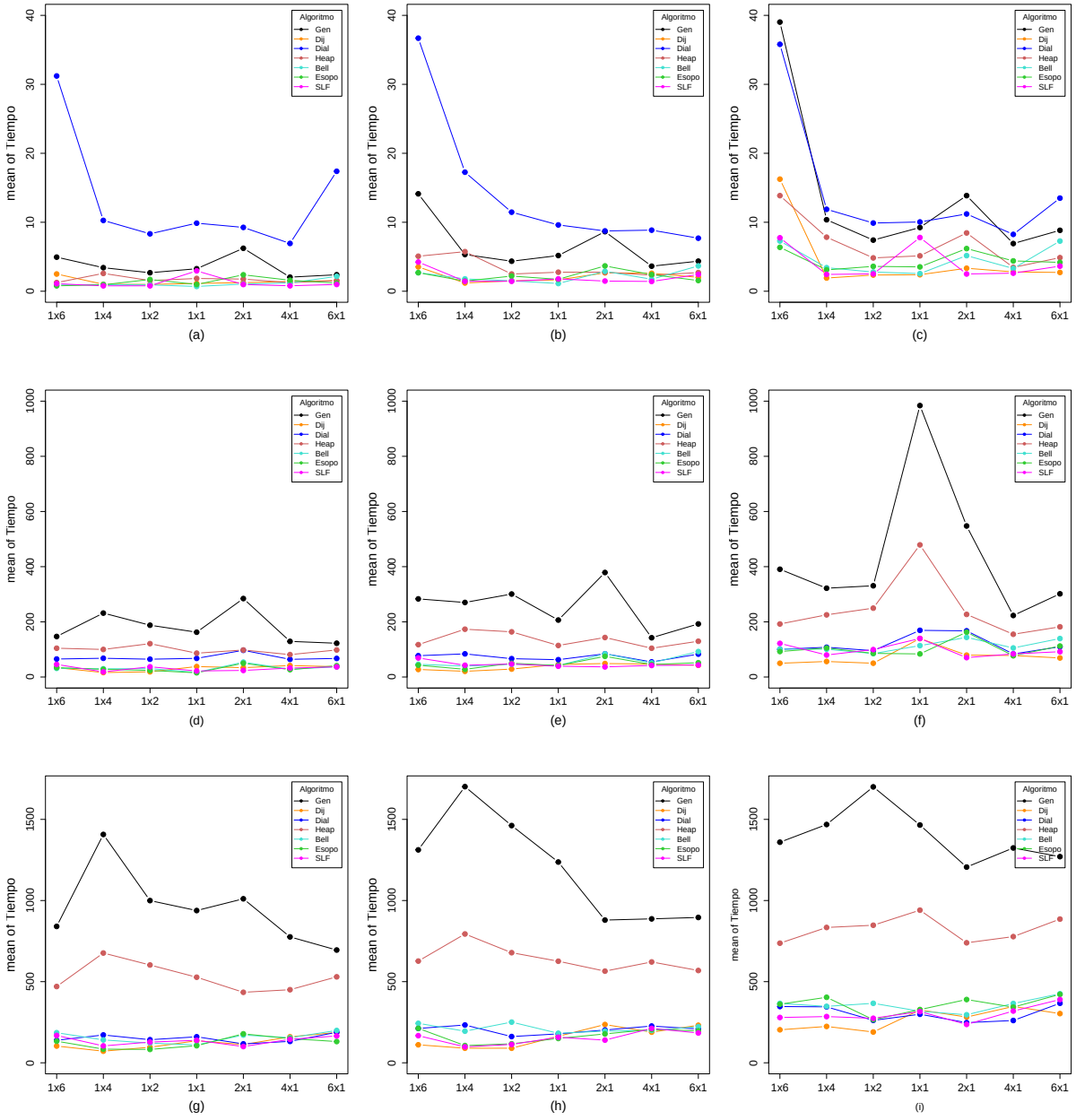

Figura 5.1: Gráfica de medias del tiempo en milisegundos de cada algoritmo estudiado por el tipo de formato. Cada fila representa la cantidad de nodos, siendo $100, 500$ y 1000 respectivamente, y cada columna representará la densidad, que serán $5, 10$ y 25.

de forma uniforme con el aumento de nodos, ya que los algoritmos de Dijkstra, Bellman-Ford, D'Esopo-Pape y SLF son los que mejores tiempos obtienen, incorporándose el algoritmo Dial a los algoritmos competitivos para redes con abundancia de nodos.

Si se contrasta el desarrollo de las gráficas a través de las columnas, es decir, teniendo en cuenta el aumento de la densidad (aumento en el número de arcos), se percibe un sutil aumento del tiempo de resolución, sobretodo para el algoritmo genérico y el Heap. Se puede apreciar que el algoritmo Dial y el algoritmo de Dijkstra pasan a ser los más competitivos para redes densas.

Por último, con respecto a la forma de las redes, es decir, el desarrollo a lo largo de cada una de las gráficas, se puede apreciar que no presenta un patrón aparente, pero para cada cantidad de nodos, da problemas una forma distinta. Para 100, la forma 1×6 tiene un tiempo de resolución mucho mayor que el resto de las formas, para 500, resaltan la forma 2×1 y 1×1 cuando su densidad aumenta, y por último, para 1000, destaca 1×4.

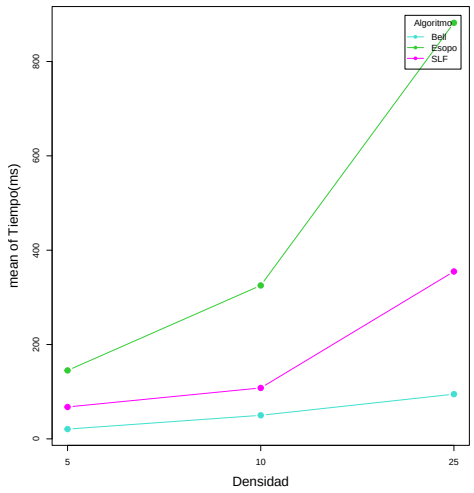

Figura 5.2: Gráfica de medias del tiempo en milisegundos de cada algoritmo estudiadas por la diferencia de densidad: $5, 10$ y 25.

Por último se ha generado una colección de problemas test en los que aparecen costos negativos, para evaluar la rapidez con la que los algoritmos detectan la existencia de ciclos negativos dentro de las redes. Por esto, solo se ha estudiado con los algoritmos que permiten este tipo de costos, a excepción del algoritmo genérico, ya que en el estudio previo se ha determinado que es el que peor resultados obtiene. Se han considerado redes de 100 nodos en forma 1×4, con densidades de $5\,\%, 10\,\%$ y $25\,\%$, en la figura 5.2 se puede observar el tiempo medio que tarda cada algoritmo en detectar la existencia de los ciclos negativos, si los hay.

Se puede apreciar que a medida que la densidad aumenta, el tiempo para detectar el ciclo lo hace también, aunque en el algoritmo de D'Esopo-Pape el crecimiento es mucho mayor. El algoritmo de Bellman-Ford es el que obtiene mejores resultados, siendo el crecimiento del tiempo muy sutil en el cambio de la densidad.

Tras el estudio realizado se puede concluir que para redes muy densas cuyos costos son positivos y con muchos nodos los algoritmos más competitivos serán el Dijkstra, Dial y SLF, mientras que para redes con pocos nodos y no densas se vuelve a recomendar el uso de los algoritmos de Dijkstra y SLF además de D'Esopo-Pape y Bellman-Ford. Es decir, para costos positivos en general se recomienda el algoritmo de Dijkstra y SLF o D'Esopo, siendo estos dos últimos modificaciones del algoritmo genérico, cuyas ideas subyacentes son heurísticas. Por otro lado para redes que contienen costos negativos, el algoritmo que mejores resultados obtendrá será el de Bellman-Ford, ya que en caso de contener ciclos, será capaz de detener el algoritmo mucho antes que los restantes.

Bibliografía

[1] Olaverri, A.G. *Teoría de Grafos*. ADD Universidad de Zaragoza, 2021.

[2] Mateo, P.M. *Investigación Operativa*. ADD Universidad de Zaragoza, 2022.

[3] Glabowski, M., Musznicki, B., Nowak, P. and Zwierzykowski, P. Review and Performance Analysis of Shortest Path Problem Solving Algorithms. *The International Journal on Advances in Software*, 7(1-2), 1993.

[4] Bertsekas, D. P. *Network Optimization: Continuous and Discrete Models*, chapter 2. Athena Scientific, 1998.

[5] Bertsekas, D. P. *Linear Network Optimization: Algorithms and Codes*, chapter 1. MIT Press, 1991.

[6] Bazaraa, M.S., Jarvis, J.J and Sherali, H.D. chapter 12. Wiley, 2010.

[7] Kershenbaum, A. A note on finding shortest path trees. *Networks*, 11(4):323–411, 1981.

[8] Shier, D.R. and Witzgall, C. Properties of Labeling Methods for Determining Shortest Path Trees. *Journal of Research of the National Bureau of Standards*, 86(3):317–330, 1981.

[9] Gallo, G. and Pallottino, S. Shortest Path Algorithms. *Annals of Operations Research*, 13:1–79, 1988.

[10] Pallottino, S. Shortest-path methods: Complexity, interrelations and new propositions. *Networks*, 14(2):257–267, 1984.

[11] Bertsekas, D.P. A Simple and Fast Label Correcting Algorithm for Shortest Paths. *Networks*, 23(8):651–709, 1993.

[12] Chen, Z. L. and Powell, W. B. A Note on Bertsekas' Small-Label-First Strategy. *Networks*, 29(2):111–116, 1997.

[13] Monk, S. *Programming the Raspberry Pi: Getting Started with Python*. McGraw-Hill/Tab Electronics, 2012.

[14] Demetrescu, C., Goldbergand, A.V. and Johnson, D.S. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Mathematical Society, 2009.

Apéndice A

Programación lineal. Dualidad

Definición 6. Un problema de programación lineal se dice que está de forma *simétrica* si todas las variables están restringidas a ser no negativas, y todas las restricciones son de tipo '$\leq$' en caso de tratarse de un problema de máximo, o de tipo '$\geq$' en caso de mínimo, es decir:

$$(P_{max})\begin{cases}\text{máx} & Z = \boldsymbol{cx}\\ \text{sujeto a:} & \boldsymbol{Ax} \leq \boldsymbol{b}\\ & \boldsymbol{x} \geq 0\end{cases} \qquad (P_{min})\begin{cases}\text{mín} & Z = \boldsymbol{cx}\\ \text{sujeto a:} & \boldsymbol{Ax} \geq \boldsymbol{b}\\ & \boldsymbol{x} \geq 0.\end{cases} \tag{A.1}$$

Dado un problema de programación lineal en forma simétrica de máximo, (P_{max}) en A.1 su problema dual asociado será el definido a continuación.

$$(D_{min})\begin{cases}\text{mín} & G = \boldsymbol{b}^T\boldsymbol{w}\\ \text{sujeto a:} & \boldsymbol{A}^T\boldsymbol{w} \geq \boldsymbol{c}^T\\ & \boldsymbol{w} \geq 0.\end{cases}$$

Las variables de holgura de los problemas (P_{max}) y (D_{min}) se agregan a las restricciones de tipo desigualdad del problema con el fin de convertirlas en una igualdad. Se denomina u a las variables de holgura del problema primal y v a las del dual.

Teorema A.1 (Condiciones de Holgura Complementaria)**.** *Dadas $\bar{\boldsymbol{x}}$ y $\bar{\boldsymbol{w}}$ soluciones factibles de los problemas simétricos de máximo y su dual, entonces $\bar{\boldsymbol{x}}$ y $\bar{\boldsymbol{w}}$ serán soluciones óptimas para sus problemas respectivos si y solo si $(\bar{\boldsymbol{w}}^T\boldsymbol{A} - \boldsymbol{c})\bar{\boldsymbol{x}} + \bar{\boldsymbol{w}}^T(\boldsymbol{B} - \boldsymbol{A}\bar{\boldsymbol{x}}) = 0$.*

Haciendo uso de las fórmulas que hay a continuación, el teorema A.1 tomará la forma del teorema 1.1.

- Sean $\bar{\boldsymbol{x}}$, $\bar{\boldsymbol{w}}$ soluciones factibles y $\bar{\boldsymbol{u}}$, $\bar{\boldsymbol{v}}$ holguras del problema primal y dual respectivamente, entonces $\bar{\boldsymbol{x}} \geq 0$, $\bar{\boldsymbol{w}} \geq 0$, $\bar{\boldsymbol{u}} \geq 0$, $\bar{\boldsymbol{v}} \geq 0$.

$$(P_{primal})\begin{cases}\text{máx} & Z = \boldsymbol{cx}\\ \text{sujeto a:} & \boldsymbol{A}\bar{\boldsymbol{x}} + \bar{\boldsymbol{u}} = \boldsymbol{b}\\ & \bar{\boldsymbol{x}} \geq 0\end{cases} \qquad (D_{dual})\begin{cases}\text{mín} & G = \boldsymbol{b}^T\bar{\boldsymbol{w}}\\ \text{sujeto a:} & \boldsymbol{A}^T\bar{\boldsymbol{w}} - \bar{\boldsymbol{v}} = \boldsymbol{c}^T\\ & \bar{\boldsymbol{w}} \geq 0\end{cases}$$

- A partir de $(\bar{\boldsymbol{w}}^T\boldsymbol{A} - \boldsymbol{c})\bar{\boldsymbol{x}} + \bar{\boldsymbol{w}}^T(\boldsymbol{B} - \boldsymbol{A}\bar{\boldsymbol{x}}) = \bar{\boldsymbol{w}}^T\bar{\boldsymbol{u}} + \bar{\boldsymbol{v}}^T\boldsymbol{x}$, se tiene $\bar{\boldsymbol{w}}^T\bar{\boldsymbol{u}} = 0$ y $\bar{\boldsymbol{v}}^T\bar{\boldsymbol{x}} = 0$ que puede expresarse de forma equivalente como:

$$\begin{cases}\bar{x}_j\bar{v}_j = 0, & j = 1, \ldots, n\\ \bar{w}_i\bar{u}_i = 0, & i = 1, \ldots, m.\end{cases}$$

La expresión sobre la cual se ha deducido esta condición, viene dada por propiedades previas de holgura complementaria, donde se afirma que si $\bar{\boldsymbol{x}}$ e $\bar{\boldsymbol{w}}$ son soluciones factibles a dichos problemas entonces $\bar{\boldsymbol{x}}$ e $\bar{\boldsymbol{w}}$ son soluciones óptimas respectivamente si y solo si verifican $(\bar{\boldsymbol{w}}^T\boldsymbol{A} - \boldsymbol{c})\bar{\boldsymbol{x}} + \bar{\boldsymbol{w}}^T(\boldsymbol{B} - \boldsymbol{A}\bar{\boldsymbol{x}}) = 0$.

Apéndice B

Problemas test y resultados

A continuación se explicará con mas detalle el generador de problemas test utilizados para el análisis del comportamiento de los algoritmos. Esta información ha sido aportada por el director del trabajo.

Esta aplicación genera una red de tipo rejilla con, inicialmente $l \times k$ nodos distribuidos en l filas y k columnas. A estos se añade un nodo previo a los nodos de la primera columna, nodo 0, y un nodo posterior a los l nodos de la última columna, nodo $l \times k + 1$.

La red genera m arcos, l desde el nodo 0 a cada uno de los nodos de la primera columna y otros desde los nodos de la última columna hasta el nodo $l \times k+1$. Para los restantes $m-2 \cdot l$ arcos, se calcula la parte entera de $(m - 2 \cdot l)/(l \cdot k)$, y ese número representarán los arcos salientes de cada uno de los nodos de la rejilla. Para que cuadre exactamente m arcos, se completará introduciendo aleatoriamente tantos arcos como sea necesario hasta llegar a los m.

Para generar los arcos que salen de cada uno de los $l \times k$ nodos de la rejilla, se toma dicho nodo asumiendo que se encuentra en la posición (i, j) de dicha rejilla. Primero se sortea si el nuevo arco va a otra columna diferente de la del nodo actual, con probabilidad p, que se tomará con un valor de 0.8; o si el arco va a un nodo dentro de la misma columna y distinta fila, con probabilidad $1 - p$, es decir, de 0.2.

Si se obtiene que el nuevo arco finaliza en una columna diferente a la del nodo actual, se genera una distribución de probabilidad que permite saltar desde la columna actual a cada una de las otras columnas de forma que la probabilidad depende de la distancia a la columna actual, ya sea hacia adelante o hacia atrás. En este caso se ha tomado que en una misma distancia, si el arco va hacia el nodo final, es decir, hacia la derecha, tiene el triple de probabilidad que si va hacia el nodo inicial. Además, al incrementar en uno la distancia, la probabilidad de salto disminuirá a la mitad. Por el contrario, si se decide que el nodo se mantiene en la misma columna, la nueva fila se sortea equiprobablemente entre el resto de filas de la columna.

Los arcos que se añaden para llegar a los m una vez introducidos los arcos que obligatoriamente salen de cada nodo de la rejilla, se generan tomando un nodo inicial aleatorio dentro de las $k - 1$ columnas y el segundo nodo se genera en una fila aleatoria que será a su vez equiprobable entre todas y una columna posterior a la del nodo seleccionado, con un salto máximo de comunas equivalente a máx$\{1, int(0.1 \cdot k)\}$.

Los costos asociados a cada arco se generan de una distribución de valores enteros y equiprobables entre dos ya marcados, ambos incluidos, que se han fijado en 1 y 100 para los problemas con costos positivos y -5 y 95 para las redes con costos negativos que se han tomado. Esta red es almacenada en un fichero en el formato *Dicmacs*.

A continuación se encuentran las tablas resúmenes de los distintos tipos de redes estudiadas, distinguiendo en la densidad y cantidad de nodos en cada una de ellas. Destacar que para cada tipo de red se han creado 30 problemas tests sobre los que se implementará el algoritmo. En cada tabla se observa el tiempo medio de resolución de cada uno de los algoritmos en milisegundos en función del formato de la red. Estas tablas representan los mismos datos que los dados en las

figuras 5.1 y 5.2, pero permiten un análisis más preciso.

Primero se tiene en cuenta las redes con todos sus costos positivos.

- Para $N = 100$
  - Para Densidad= 5.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	4.9433833	3.4138467	2.6828633	3.2614500	6.2289933	2.0421967	2.3872333
Dij	2.4975833	1.0076167	0.9848600	1.1361100	1.2447033	1.4181200	1.5393533
Dial	31.2120867	10.2679567	8.3245633	9.8686567	9.2561100	6.9307567	17.3926300
Heap	1.2535033	2.5850767	1.5511700	1.8510900	1.7617600	1.2421967	1.5392467
Bell	0.8908433	0.9614400	0.9212033	0.6964067	1.0394400	1.2004667	2.1544167
Esopo	0.7825833	0.9391867	1.7036733	0.9820000	2.3795367	1.5925900	1.1816000
SLF	1.0910867	0.7840067	0.7864600	2.9531867	0.9754067	0.7910133	0.9825033

Tabla B.1: Tabla estadística que representa la media del tiempo en milisegundos para $N = 100$ y Densidad= 5 %.

  - Para Densidad= 10.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	14.121550	5.323403	4.352643	5.173990	8.637717	3.634133	4.358157
Dij	3.528117	1.195700	1.513000	1.645350	2.668463	2.603463	2.012403
Dial	36.691620	17.259853	11.463627	9.612643	8.742570	8.856943	7.690390
Heap	5.076380	5.737680	2.497570	2.757870	2.763697	2.326600	2.700243
Bell	2.694350	1.790510	1.511783	1.118033	2.891327	1.738577	3.672987
Esopo	2.685067	1.471770	2.197087	1.729927	3.670133	2.412363	1.554737
SLF	4.232260	1.439470	1.439173	1.784423	1.475430	1.417370	2.430053

Tabla B.2: Media del tiempo en milisegundos para $N = 100$ y Densidad= 10 %.

  - Para Densidad= 25.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	39.024363	10.362870	7.415030	9.243660	13.863080	6.913957	8.831767
Dij	16.252230	1.917113	2.349630	2.398047	3.332830	2.803260	2.724463
Dial	35.802567	11.880840	9.889510	10.041813	11.204020	8.241577	13.497870
Heap	13.860167	7.836580	4.833993	5.131047	8.451633	3.453290	4.869013
Bell	7.280280	3.414790	2.786890	2.556047	5.164390	3.297217	7.276727
Esopo	6.367487	3.097807	3.603093	3.530517	6.183033	4.399310	4.184737
SLF	7.746573	2.438193	2.510293	7.787557	2.502187	2.618727	3.638220

Tabla B.3: Media del tiempo en milisegundos para $N = 100$ y Densidad= 25 %.

- Para $N = 500$
  - Para Densidad= 5.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	146.87948	231.71308	187.75016	162.42901	284.36810	129.17379	122.36780
Dij	34.02096	16.20041	18.93551	37.88086	33.83091	41.55234	37.40422
Dial	65.26186	67.94711	64.56541	67.65150	96.80110	64.09599	67.33806
Heap	104.37202	99.88700	120.93618	86.36102	97.98055	81.20279	98.01178
Bell	31.41976	29.72975	30.49598	16.76068	53.73594	28.25483	36.96791
Esopo	34.05244	27.88661	23.85529	15.29160	49.90568	26.67922	40.96178
SLF	45.62452	18.66935	37.07670	22.01906	23.85098	32.43326	36.83684

Tabla B.4: Media del tiempo en milisegundos para $N = 500$ y Densidad= 5 %.

- Para Densidad= 10.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	283.17220	270.24234	300.75959	206.49235	378.78770	142.32770	192.42838
Dij	27.38667	20.74221	28.74934	45.66189	48.85034	46.91512	42.62307
Dial	77.27568	83.85945	66.43727	62.76419	84.14583	54.69922	82.98500
Heap	117.33630	173.31925	163.75290	114.31767	143.20863	104.29336	129.62983
Bell	46.00560	39.55882	46.10512	41.97179	83.66775	51.29028	92.24667
Esopo	42.75253	27.92060	49.94408	41.44378	74.90775	45.37160	51.71502
SLF	69.06445	42.59004	47.17327	39.16109	36.70338	41.92610	43.37764

Tabla B.5: Media del tiempo en milisegundos para $N = 500$ y Densidad= 10 %.

- Para Densidad= 25.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	390.60683	321.92775	330.74715	983.94711	547.51783	222.99194	301.51632
Dij	49.89059	56.27866	50.00214	139.59553	79.29547	77.98803	69.07425
Dial	100.17582	107.86998	95.48505	169.15405	167.36098	83.59601	109.67623
Heap	192.22355	225.36473	249.65763	479.00798	227.00526	154.93527	181.91381
Bell	101.23848	100.83735	84.95960	113.75162	143.69078	105.33049	139.40830
Esopo	92.37939	105.23087	85.68959	83.66933	162.56531	77.09639	112.24787
SLF	121.85680	79.81055	99.35306	139.61136	70.27395	83.72806	91.72396

Tabla B.6: Media del tiempo en milisegundos para $N = 500$ y Densidad= 25 %.

- Para $N = 1000$

  - Para Densidad= 5.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	840.4164	1407.12187	999.49124	938.1866	1010.5278	775.8545	694.8643
Dij	103.2384	71.58635	95.88300	135.7490	112.4831	160.2048	186.5426
Dial	138.7218	171.40885	142.18029	160.1962	116.4923	131.8441	187.9273
Heap	470.0631	676.44435	602.64390	527.3678	434.5030	450.3304	529.6186
Bell	185.4344	140.95666	121.37230	110.3620	170.8471	154.6204	199.0977
Esopo	133.1537	84.16786	82.40359	105.1826	178.4629	149.2580	130.9520
SLF	168.5073	104.07238	127.59259	138.3905	100.1240	144.5655	165.4383

Tabla B.7: Media del tiempo en milisegundos para $N = 1000$ y Densidad= 5 %.

- Para Densidad= 10.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	1312.1108	1701.73187	1461.87461	1237.0762	879.4927	886.9374	895.6284
Dij	110.8332	89.87194	89.78103	163.5117	235.7451	188.9447	231.2931
Dial	211.6824	232.89590	161.62963	178.2123	202.0085	226.3554	208.2938
Heap	626.8365	793.81802	678.52003	625.9275	564.7822	621.4377	568.7393
Bell	243.1595	194.82609	250.66439	181.8565	195.3071	199.3022	222.4877
Esopo	212.1692	106.18269	117.02600	149.8923	177.9223	205.7446	196.9558
SLF	167.0752	98.15223	113.13559	157.5725	139.7641	212.0660	183.7871

Tabla B.8: Media del tiempo en milisegundos para $N = 1000$ y Densidad= 10 %.

- Para Densidad= 25.

	Forma						
Algoritmo	1×6	1×4	1×2	1×1	2×1	4×1	6×1
Gen	1359.3315	1468.4184	1700.3284	1465.1374	1206.1127	1323.9907	1270.3655
Dij	203.3756	223.9715	189.7181	325.3818	281.8745	347.0623	303.1404
Dial	346.6228	344.6434	261.4775	299.0959	247.4207	260.6279	366.6635
Heap	737.0820	833.8920	847.6045	940.4855	739.4373	777.5109	885.4802
Bell	365.8079	347.6245	366.5095	317.7303	295.8496	365.4115	425.3093
Esopo	361.0376	403.5775	265.6948	327.8105	389.8034	342.8026	421.7340
SLF	279.1620	284.8404	274.2223	315.6332	236.4630	318.9357	389.5721

Tabla B.9: Media del tiempo en milisegundos para $N = 1000$ y Densidad= 25 %.

A continuación se tiene la tabla para el estudio de las redes con costos negativos, que representan el tiempo medio que tarda cada algoritmo en detectar un ciclo en la red dada.

	Densidad		
Algoritmo	5	10	25
Bell	20.73292	49.93673	94.7648
Esopo	145.08446	325.24244	882.0578
SLF	67.65166	108.00250	354.8277

Tabla B.10: Media del tiempo en milisegundos para $N = 100$ y Forma= 1×4.

Apéndice C

Implementación de los algoritmos

En este apéndice se pueden encontrar la implementación de los algoritmos explicados previamente, y el lector de ficheros Dicmacs programados con el lenguaje *Python*.

C.1. Lectura ficheros Dicmacs

Función que lee los datos de un archivo en formato Dicmacs[14].

```
import numpy as np

def datos(nombre_archivo):
    try:
        #Se inicializan el contador de arcos, y la cantidad total de arcos y nodos
        contador=0
        marcos=0
        nodos=0

        #Abre y lee el archivo de texto. Lee cada línea, si está vacía o con espacio,
        #continua sino guarda la primera letra.
        with open (nombre_archivo,'r') as archivo:
            for linea in archivo:
                if not linea.strip():
                    continue
                primera_letra = linea[0]

                #Si la letra es 'p', se guarda el número total de nodos y arcos y se crean
                #los vectores donde se guardarán el origen, destino y costo de los nodos.
                if primera_letra == 'p':
                    l=linea.split()
                    nodos=int(l[2])
                    marcos=int(l[3])
                    start=np.zeros(marcos).astype(int) #pasa de variable a número
                    end=np.zeros(marcos).astype(int)
                    c=np.zeros(marcos).astype(int)

                #Si la letra es 'a', se guarda la información de los arcos en los vectores
                #especializados para ello.
                if primera_letra == 'a':
                    l=linea.split()
                    start[contador]=int(l[1])
```

```
                    end[contador]=int(l[2])
                    c[contador]=int(l[5])
                    contador=contador+1
                else:
                    continue

        #Se crean las listas de adyacencia y se añaden los arcos correspondientes
        #de cada nodo.
        IN=[]
        OUT=[]
        for i in range (nodos):
            IN.append([])
            OUT.append([])
        for i in range (contador):
            IN[end[i]].append(i)
            OUT[start[i]].append(i)

    except FileNotFoundError:
        print("El archivo '{}' no se encontró.".format(nombre_archivo))

    #Se devuelven las siguientes variables cuando se llame a la función.
    return IN,OUT,c,marcos,nodos,start,end
```

C.2. Algoritmo Genérico

Programa que ejecuta el algoritmo genérico explicado en el Capítulo 2 y devuelve por pantalla las distancia mínima que hay del nodo inicial a cada uno de los nodos pertenecientes al grafo y el tiempo de ejecución de dicho programa.

```
import sys
import numpy as np
from math import inf
import random
import time

#Se carga la función 'datos', guardada en el fichero 'funciondatos', y se llama a
#dicha función, que leerá los datos del fichero de texto que se meta por pantalla.
import funciondatos
iname=input('Nombre del fichero de datos: ')
(IN,OUT,c,marcos,nodos,start,end)=funciondatos.datos(iname)

#Variable que indica cuando empieza a contar el tiempo.
inicio = time.perf_counter_ns()
#Se define V como una lista vacía, a la que se va añadiendo e eliminando elementos durante
#el programa. Se inicializa V={0}.
V=[]
V.append(0)
#Se define d como el vector de etiquetas. Se inicializa el vector, siendo todos sus valores
#igual a infinito menos el del nodo inicial, que será 0.
d=[inf]*nodos
d[0]=0
```

```
#Se define M, constante para detectar la presencia de ciclos negativos.
M=min(-1,(nodos-1)*min(c))

#Se hacen iteraciones mientras la longitud del vector V sea mayor que 0.
while len(V)>0:
    i=V.pop(random.randint(0, len(V)-1)) #elimina un nodo aleatorio de V.

    #Para cada arco saliente del nodo eliminado se comprueban las CHC:
    for a in OUT[i]:
        if d[end[a]]>d[start[a]]+c[a] and end[a]!=0:
            d[end[a]]=d[start[a]]+c[a]
            #Detecta la presencia de un ciclo y para el algoritmo.
            if min(d)<M:
                sys.exit()
            #Se añade el nodo a la lista si no estaba ya en ella.
            if end[a] not in V:
                V.append(end[a])

fin = time.perf_counter_ns()
print('Las distancias mínimas del nodo 1 a todos los demás son:',d,'\n',
      'El tiempo de ejecución fue:', (fin - inicio)/1000000 ,'milisegundos')
```

C.3. Algoritmo de Dijkstra

Programa que ejecuta el algoritmo de Dijkstra descrito en el Sección 3.1 y devuelve por pantalla las distancia que hay del nodo inicial a cada uno de los nodos pertenecientes al grafo.

En este programa y en los posteriores se explicarán solamente las partes que se han alterado, el resto de aclaraciones han sido omitidas para evitar la repetición. Por esta razón también se ha prescindido de medir el tiempo de ejecución, en caso de desear hacerlo, se utilizarán el procedimiento realizado en el algoritmo genérico.

```
import sys
import numpy as np
from math import inf

import funciondatos
iname=input('Nombre del fichero de datos:')
(IN,OUT,c,marcos,nodos,start,end)=funciondatos.datos(iname)

#comprueba si existe algun c_ij >=0, si es así se para el algoritmo.
if  c.any()<0:
    sys.exit()
V=[]
V.append(0)
d=[inf]*nodos
d[0]=0

while len(V)>0:
    #se inicializa el mínimo antes del bucle, se escoje el la menor etiqueta de los nodos
    #de V y se elimina dicho nodo de V j de V.
    minimo=inf
```

```
    for j in V:
        if d[j] < minimo:
            minimo=d[j]
            sale=j
    V.remove(sale)

    for a in OUT[sale]:
        if d[end[a]]>d[start[a]]+c[a] and end[a]!=0:
            d[end[a]]=d[start[a]]+c[a]
            if end[a] not in V:
                V.append(end[a])
print ('Las distancias mínimas del nodo 1 a todos los demás son:',d)
```

C.4. Algoritmo de Dijkstra con cola de prioridad

Programa que ejecuta el algoritmo de Dijkstra con colas de prioridad detallado en el Sección 3.2 y devuelve por pantalla las distancia que hay del nodo inicial a cada uno de los nodos pertenecientes al grafo.

```
import sys
import heapq
import numpy as np
from math import inf

import funciondatos
iname=input('Nombre del fichero de datos: ')
(IN,OUT,c,marcos,nodos,start,end)=funciondatos.datos(iname)

if  c.any()<0:
    sys.exit()
d=[inf]*nodos
d[0]=0
#Se inicializa y define la pila como una lista de tuplas, que tendrán como segundo elemneto
#los nodos de V, y como primero sus etiquetas.
pila=[(0,0)]

#Se hacen iteraciones mientras que la longitud de la pila sea mayor que 0 (ya que V es el
#conjunto de los segundos componentes de las tuplas de la pila).
while len(pila)>0:
    #Elimina de la pila la tupla con el primer elemento más pequeño (nodo con menor etiqueta).
    D,i=heapq.heappop(pila)

    for a in OUT[i]:
        if d[end[a]]>d[start[a]]+c[a] and end[a]!=0:
            d[end[a]]=d[start[a]]+c[a]
            #Se comprueba que el nodo eliminado de la pila no pertenezca a V y en ese
            #caso se añade.
            if all (tupla[1]!=end[a] for tupla in pila):
                heapq.heappush(pila,(d[end[a]],end[a]))
print ('Las distancias mínimas del nodo 1 a todos los demás son:',d)
```

C.5. Algoritmo Dial

Programa que ejecuta el algoritmo Dial explicado en la Sección 3.3 y devuelve la distancia mínima del nodo inicial a todos los nodos de la red.

```
import sys
import numpy as np
from math import inf
import funciondatos
iname=input('Nombre del fichero de datos: ')
(IN,OUT,c,marcos,nodos,start,end)=funciondatos.datos(iname)
#comprueba si existe algun c_ij>=0 o no entero, si es así se para el algoritmo

if  c.any()<0 or (all(isinstance(x, np.int32) for x in c))==False:
    sys.exit()
V=[]
V.append(0)
d=[inf]*nodos
d[0]=0
#Se inicializan los buckets, la cota y el contador de índices de buckets.
C=(nodos-1)*max(c)
B=[[]for _ in range(C+1)]
B[0].append(0)
indice=0

while len(V)>0:
    #Si el bucket contiene algún nodo, para y examina, si está vacío pasa al sigueinte.
    #Si ya ha examinado todos, para el bucle.
    while len(B[indice])==0:
        indice+=1
    if indice==C+1:
        break
    #se elimina un nodo del bucket (da igual cual ya que todos tienen la misma etiqueta)
    #y a su vez de V.
    sale=B[indice].pop(0)

    V.remove(sale)
    for a in OUT[sale]:
        if d[end[a]]>d[start[a]]+c[a] and end[a]!=0:
            #si la distancia no es infinito, está contenido en otro bucket, por lo que
            #habrá que eliminarlo previamente para después poder añadirlo.
            if d[end[a]]!=inf:
                B[d[end[a]]].remove(end[a])
            #Se actualiza la distancia y se añade el nodo al bucket.
            d[end[a]]=d[start[a]]+c[a]
            B[d[end[a]]].append(end[a])
            if end[a] not in V:
                V.append(end[a])
print ('Las distancias mínimas del nodo 1 a todos los demás son:', d)
```

C.6. Algoritmo de Bellman-Ford

Se trata de un programa que ejecuta el algoritmo de Bellman-Ford, detallado en la Sección 4.1 del trabajo. Este programa devuelve la distancia mínima entre el nodo inicial tomado y el resto de nodos pertenecientes a la red.

```
import sys
import numpy as np
from math import inf

import funciondatos
iname=input('Nombre del fichero de datos: ')
(IN,OUT,c,marcos,nodos,start,end)=funciondatos.datos(iname)

V=[]
V.append(0)
d=[inf]*nodos
d[0]=0

#Se hacen tantas iteraciones como nodos hay en el grafo
for i in range(nodos):
    it=0
    long=len(V)
    #Se hacen tantas iteraciones como nodos había en V al final de la iteración anterior
    while it< long:
        it+=1
        j=V.pop(0)

        for a in OUT[j]:
            if d[end[a]]>d[start[a]]+c[a]:
                #Si en la iteracón N cambia, existe un ciclo negativo
                if (i==nodos-1):
                    print('En el grafo hay al menos un ciclo negativo')
                    sys.exit()
                d[end[a]]=d[start[a]]+c[a]
                if end[a] not in V:
                    V.append(end[a])
print ('Las distancias mínimas del nodo 1 a todos los demás son:', d)
```

C.7. Algoritmo D'Esopo-Pape

Programa que desarrolla el algoritmo especificado en la Sección 4.2, es decir, el D'Esopo Pape, devolviendo la distancia mínima entre el nodo inicial y el resto de ellos.

```
import sys
import numpy as np
from math import inf

import funciondatos
iname=input('Nombre del fichero de datos: ')
```

```
(IN,OUT,c,marcos,nodos,start,end)=funciondatos.datos(iname)

V=[]
V.append(0)
d=[inf]*nodos
d[0]=0

while len(V)>0:
    #coge un elemento de la lista y lo elimina en orden FIFO
    i=V.pop(0)

    for a in OUT[i]:
        if d[end[a]]>d[start[a]]+c[a] and end[a]!=0:
            if end[a] not in V:
                if d[end[a]]==inf:
                    V.append(end[a])#lo añade al final de la lista
                else:
                    V.insert(0,end[a])#añade en el elemnto V[0] el end[a]
            d[end[a]]=d[start[a]]+c[a]
print ('Las distancias mínimas del nodo 1 a todos los demás son:', d)
```

C.8. Algoritmo SLF

Programa que detalla el algoritmo SLF, explicado en la Sección 4.3, y devuelve la distancia mínima entre el nodo inicial y el resto de ellos.

```
import numpy as np
from math import inf
import time

import funciondatos
iname=input('Nombre del fichero de datos: ')
(IN,OUT,c,marcos,nodos,start,end)=funciondatos.datos(iname)

V=[]
V.append(0)
d=[inf]*nodos
d[0]=0

while len(V)>0:
    i=V.pop(0)

    for a in OUT[i]:
        if d[end[a]]>d[start[a]]+c[a] and end[a]!=0:
            d[end[a]]=d[start[a]]+c[a]
            #Si V está vacío se añade el nodo.
            if len(V)==0:
                V.append(end[a])
            #Si no está vacío, se mira si la etiqueta del nodo es mayor que la del primer
            #nodo de V
            elif end[a] not in V:
```

```
                if d[end[a]]>d[V[0]]:
                    V.append(end[a]) #lo añade al final de la lista
                else:
                    V.insert(0,end[a]) #añade en el nodo V[0] el end[a]
print ('Las distancias mínimas del nodo 1 a todos los demás son:', d)
```