Algoritmos para el problema de flujo a costo mínimo

Rebeca Cantín Sánchez

Trabajo de fin de grado en Matemáticas
Universidad de Zaragoza

Director del trabajo: Pedro Mateo Collazos
4 de diciembre de 2020

Resumen

Network flows are a problem model that is on the cusp of several fields of research, including computer science, applied mathematics, engineering, management, and operations research.

In this report, we will focus on the field of Operational Research, more specifically, on the Minimum Cost Flow Problem (MCFP). This is one of the most important network flow problems, whose purpose of determining the best way to send a merchandise through a network with lower cost.

This model has a number of well-known applications: the distribution of a product from factories to warehouses, or from warehouses to retailers; the flow of raw materials and intermediate goods through several machining stations on a production line; the route of cars through a network of urban streets; and the route of calls through the telephone system, water flows in the distribution system or in the sewerage, etc.

In particular, it relies on graph theory to develop MCFP, using directed graphs. As a purpose of these problems, a network is created that seeks to satisfy the greatest number of node demands, using for it the excess of available supplies in other nodes.

Throughout this work, two types of algorithms are developed to solve said model, the primal-dual and out-of-kilter algorithms. Both use a similar algorithmic strategy, in each iteration they solve a shorter route problem and increase the flow along one or more shorter routes. However, they vary in their tactics. The primal-dual algorithm uses a maximum flow calculation to increase flow along several shorter routes simultaneously. Instead, the Out-of-Kilter algorithm allows arc flows to cross their flow limits, using shorter route calculations to find flows that meet flow limits and optimality conditions.

Therefore, the work is structured in 5 parts.

In the first chapter, the basic concepts of graph theory and the theory needed to develop both MCFP algorithms are presented. In addition, duality theory is explained, thus each programming problem can be associated with another problem called dual problem. It should be noted that in the latter part, its various conditions of optimality are explained, very important for the explanation of both algorithms.

In the second chapter, the Primal-Dual algorithm is developed. To do this, a pseudoflow is used and the transformation of the MCFP into a problem with a single node excess and a single node defect is explained. A source node and a sink node must be added to our network, where each of them has the total positive and negative capacity of our nodes, respectively. Afterwards, the algorithm is presented in detail and to finish, a practical example is explained where we can observe step by step the application of this algorithm.

In the third chapter, the Out-Of-Kilter algorithm is developed. As in the second chapter, we have to add a source node and a sink node to your network, with the difference that they join with an arc forming a cycle. Subsequently, the algorithm is explained in detail and a practical example is explained

where the application of this algorithm is observed step by step.

In the fourth chapter, the implementation of these algorithms in the *Python* language is explained and the comparison of both is detailed using tables made from the data obtained from the selected examples. Finally, a brief conclusion is made about the results obtained.

Índice general

Capítulo 1

Introducción

Los problemas de flujo de red son uno de los problemas de optimización más importantes y frecuentes. Surgen en el análisis y diseño de grandes sistemas, como las redes de comunicación, transporte y fabricación. Aunque también se pueden utilizar para modelar importantes problemas combinatorios, como la asignación, el camino más corto y los problemas de los vendedores ambulantes.

En particular, el problema de flujo de costo mínimo (PFCM) es uno de los problemas de flujo de red más importantes, podemos definirlo como la mejor manera de determinar el envío de una mercancía a través de una red con el menor costo. Con este algoritmo buscamos satisfacer la mayor cantidad de demandas de los nodos, utilizando para ello el exceso de suministros disponibles en otros nodos.

El PFCM modela numerosos e importantes problemas reales, como por ejemplo la ruta de automóviles a través de una red de calles urbanas; y el itinerario de las llamadas a través del sistema telefónico, flujos de aguas en sistema de distribución o en alcantarillado.

Hay una gran cantidad de algoritmos que resuelven este problema. En este trabajo, vamos a desarrollar dos de ellos, los algoritmos primal-dual y out-of-kilter. Ambos son muy similares, ya que en cada iteración resuelven un problema de ruta más corta y aumentan el flujo a lo largo de una o más rutas más cortas. Sin embargo, las tácticas utilizadas cambian.

El primero determina nodos que tienen exceso y defecto, y calculan una ruta de costo mínimo para enviar flujo de uno a otro reduciendo tanto el exceso como el defecto. El proceso se repite hasta conseguir que todos los nodos tengan exceso y defecto nulos. El segundo, para cada arco define un elemento denominado número de 'kilter' que mide el grado de infactibilidad provocada por dicho arco. El algoritmo itera seleccionando dicho arco y construyendo trayectorias entre los extremos de dichos arcos que permiten reducir su infactibilidad.

El hecho de que podamos implementar algoritmos iterativos de tantas maneras demuestra la versatilidad que tenemos para resolver problemas de flujo de costo mínimo. Dado que los problemas de flujo de costo mínimo son programas lineales, nos permite utilizar herramientas de programación lineal para resolver estos problemas. De hecho, nos podemos basar en diversos conceptos que hemos estudiado en la asignatura de Investigación Operativa del grado, como son, la dualidad y las condiciones de optimalidad en programación lineal.

Para desarrollar estos algoritmos correctamente, primero vamos a establecer la notación y algunas definiciones que usamos a lo largo de nuestra discusión. Posteriormente explicaremos ambos algoritmos y realizaremos un análisis de diversos ejemplos empleándolos. Finalmente, implementaremos los algoritmos y haremos una comparación exhaustiva para poder llegar a una conclusión final.

1.1. Definiciones

En esta sección se presentan las definiciones de la teoría de grafos [6] sobre los que se trabajará y algunos elementos relacionados.

• **Definición 1.** Un **arco** viene dado por un elemento que une dos puntos distintos, a los que llamaremos **nodos o vértices**. En nuestro trabajo, indica un canal por el que podemos enviar un ítem desde un nodo a otro. Hay distintos tipos de arcos. Un **arco no dirigido** (i, j) puede ser considerado como una doble calle con transito permitido en ambas direcciones (tanto desde nodo i al nodo j o desde el nodo j al nodo i). Por otra parte, un **arco dirigido** (i, j) se comporta como una calle de una dirección y permite circular solamente desde el nodo i al nodo j.

• **Definición 2.** La **capacidad** es la cantidad máxima de flujo que puede soportar cada arco.

• **Definición 3.** Un **grafo** $G = (N,A)$ consiste en un conjunto finito de nodos o vértices N y un conjunto $A = \{(i, j)|i, j \in N\} \subseteq N \times N$ de arcos. Hay distintos tipos de grafos. Un **grafo dirigido** $G = (N,A)$ es un grafo en el que existen arcos dirigidos. Un **grafo no dirigido** $G = (N,A)$ es un grafo en el que todos sus arcos son no dirigidos.

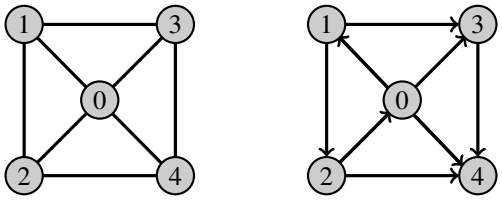


Figura 1.1: **Grafo no dirigido y grafo dirigido**

• **Definición 4.** Una **red** es un grafo cuyos nodos y/o arcos tienen asociado un valor numérico.

• **Definición 5.** Un grafo $G' = (N',A')$ es un **subgrafo** de $G = (N,A)$ si $N' \subseteq N$ y $A' \subseteq A$. Decimos que $G' = (N',A')$ es un subgrafo de G inducido por N' si A' contiene a cada arco de A con ambos extremos en N'.

• **Definición 6.** Un **camino** en un grafo dirigido $G = (N,A)$ es un subgrafo de G que consiste en una secuencia de nodos $i_1 - i_2 - ... - i_{r-1} - i_r$ y arcos asociados $a_1 - a_2 - ... - a_{r-1} - i_r$ que satisfacen que para todo $1 \leq k \leq r-1$, tenemos $a_k = (i_k, i_{k+1}) \in A$ o $a_k = (i_{k+1}, i_k) \in A$.

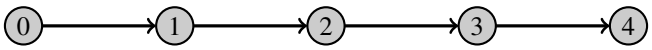


Figura 1.2: **Camino.**

• **Definición 7.** Una **circulación** o *ciclo* es un camino con el primer y último nodos iguales. Se denota de la manera siguiente $i_1 - i_2 - ... - i_r - i_1$.

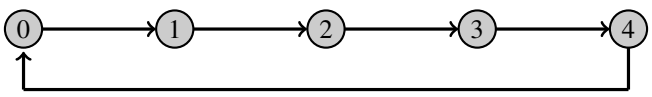


Figura 1.3: **Circulación o ciclo.**

1.2. Problema de Flujo de Costo Mínimo

En este apartado vamos a presentar el Problema de Flujo de Costo Mínimo, PFCM. El formato de este puede variar dependiendo de la utilidad final o del planteamiento del estudio del problema. En este trabajo, vamos a partir de varias suposiciones esenciales y una notación especifica para el estudio que vamos a realizar.

1.2.1. Definición del PFCM y conceptos previos

Sea $G = (N,A)$ una red dirigida. El objetivo del PFCM es encontrar un conjunto de variables de decisión que minimicen una función de costo lineal. Estas variables las llamamos *flujos*, se denotan mediante x_{ij} para cada arco $(i, j) \in A$. Además, a cada uno de estos arcos se le asocia un costo c_{ij}, que indica el costo por unidad de flujo que atraviesa dicho arco. Por otra parte, hay que tener en cuenta que el flujo de cada arco del problema está sujeto a restricciones. Por lo que a cada arco se le asocia también una limitación en el valor de su flujo.

Podemos definir el problema de flujo de Costo Mínimo de la siguiente forma:

$$\text{Minimizar} \quad z(x) = \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1.2.1a}$$

sujeto a

$$\sum_{\{\, j:(i,j)\in A \,\}} x_{ij} - \sum_{\{\, j:(j,i)\in A \,\}} x_{ji} = b(i) \quad \forall i \in N \tag{1.2.1b}$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \quad \forall i \in N \tag{1.2.1c}$$

(1.2.1a) representa el costo total de enviar los diversos flujos a través de los arcos.

El primer tipo de restricciones (1.2.1b), se denominan *restricciones de balance de masa* o restricción de conservación de flujo. El primer término representa la *salida* total de flujo del nodo mientras que el segundo representa la *entrada* total de flujo del nodo. La restricción del balance de masa establece que la salida menos la entrada debe ser igual a la *oferta/demanda* del nodo. Definimos la *oferta/demanda* de un nodo $i \in N$, representado por el numero entero $b(i)$ para cada arco. Si $b(i) > 0$, el nodo i es un *nodo con oferta*, si $b(i) < 0$, el nodo i es un *nodo con demanda*, con una demanda de $-b(i)$. Finalmente, si $b(i) = 0$, se dice que el nodo i es un *nodo de transbordo*.

Respecto a las segundas restricciones (1.2.1c), nos referimos a ellas como *restricciones de cota de flujo* (o simplemente restricciones de 'cota'). Los límites de flujo sirven para limitar la cantidad de flujo que puede fluir por cada arco. Podemos diferenciar dos tipos de cotas: limite inferior l_{ij} que designa la cantidad mínima de flujo que debe circular por el arco y límite superior u_{ij}, la cantidad máxima que debe circular.

Aunque este es el problema de flujo de costo mínimo original, existen diversas variaciones que son equivalentes a este. Dependiendo del contexto estas variantes pueden ser más convenientes que las otras. Algunas de ellas son, el problema con cotas inferiores nulas o con cotas superiores nulas, reformular el problema como un problema de asignación o de circulación, etc, En esta memoria para el algoritmo del capítulo 2, vamos a emplear la variante del PFCM estableciendo las cotas inferiores nulas.

Si nuestro problema ya tiene las cotas inferiores nulas no es necesario hacer nada, en otro caso si $l_{ij} \neq 0$ debemos reformular nuestro problema, para ello remplazamos x_{ij} por $x_{ij} - l_{ij}$ y ajustamos las cotas superiores y la oferta/demanda de acuerdo a la nueva variable de decisión, siendo

$$u_{ij} := u_{ij} - l_{ij} \tag{1.2.2a}$$

$$b(i) := b(i) - \sum_{\{\, j:(i,j)\in A \,\}} l_{ij} + \sum_{\{\, j:(j,i)\in A \,\}} l_{ji} \tag{1.2.2b}$$

Posteriormente, tras la resolución del problema si este es factible, los flujos óptimos y el valor óptimo del problema original se obtienen mediante la suma de l_{ij} al flujo óptimo de cada arco (i,j) y sumando $\sum_{(i,j)\in A} c_{ij} l_{ij}$ al valor óptimo del problema transformado.

Entonces el problema quedaría de la forma:

$$\text{Minimizar} \quad z(x) = \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{1.2.3a}$$

sujeto a

$$\sum_{\{\, j:(i,j)\in A \,\}} x_{ij} - \sum_{\{\, j:(j,i)\in A \,\}} x_{ji} = b(i) \quad \forall i \in N \tag{1.2.3b}$$

$$0 \le x_{ij} \le u_{ij} \quad \forall i \in N \tag{1.2.3c}$$

En lo que sigue, asumiremos que los problemas serán factibles.

A continuación, exponemos algunas nociones y suposiciones que debemos aclarar antes de explicar los algoritmos que queremos ver.

Pseudoflujo

El *pseudoflujo* es una función $x : A \longrightarrow R^+$ que satisface únicamente las restricciones de capacidad y no negatividad, no necesita que satisfaga las restricciones de balance de masa, de (1.2.1) o (1.2.3).

Red Residual

Un concepto imprescindible dentro de la teoría del PFCM es la *red residual* $G(x)$, esta está asociada a un flujo o pseudoflujo x definido sobre la red G de la siguiente manera. Para cada arco $(i,j) \in A$ consideramos dos arcos (i,j) y (j,i). Donde el arco (i,j) tiene costo c_{ij} y *capacidad residual* $r_{ij} = u_{ij} - x_{ij}$, y el arco (j,i) tiene costo $c_{ji} = -c_{ij}$ y capacidad residual $r_{ji} = x_{ij}$. La red residual contendrá únicamente los arcos que acabamos de definir con capacidad residual positiva. Los arcos de la red residual son arcos que indican que es posible enviar/devolver flujo del nodo i al nodo j.

Por comodidad y simplicidad en los desarrollos, realizaremos, sin pérdida de generalidad, las siguientes suposiciones:

Suposición 1.1
Todos los parámetros empleados en el PFCM son enteros.

Suposición 1.2
Las ofertas/demandas de los nodos satisfacen la condición de $\sum_{i\in N} b(i) = 0$ *y el PFCM tiene al menos una solución factible.*

Suposición 1.3
Todos los costes de los arcos son no negativos.

1.2.2. Dualidad

A cada problema de programación lineal (o problema primal) le podemos asociar otro problema de programación lineal íntimamente relacionado con este problema, llamado su *problema dual*. Vamos a enunciar y probar los resultados de la teoría de la dualidad para el PFCM, que explican la relación entre el problema primal y su dual y que nos permitirán establecer conclusiones de optimalidad sobre ambos problemas.

Para construir el problema dual, asociamos una *variable dual* con cada restricción del primal, excepto la restricción de no negatividad en los flujos de arco. Para el PFCM indicado en (1.2.3), asociamos la variable $\pi(i)$ con la restricción de balance de masa del nodo i y la variable α_{ij} con la restricción de capacidad de arco (i,j). A partir de estas variables, el *problema de flujo de costo mínimo dual* se puede establecer de la siguiente manera:

$$\text{Maximizar} \quad w(\pi,\alpha) = \sum_{i\in N} b(i)\pi(i) - \sum_{(i,j)\in A} u_{ij}\alpha_{ij} \tag{1.2.4a}$$

sujeto a

$$\pi(i) - \pi(j) - \alpha_{ij} \le c_{ij} \quad \forall (i,j) \in A \tag{1.2.4b}$$

$$\alpha_{ij} \ge 0 \quad \forall (i,j) \in A \quad \text{y} \quad \pi(j) \quad \text{no restringido} \quad \forall i \in N \tag{1.2.4c}$$

Para reescribir el problema dual, debemos eliminar las variables duales α_{ij} de la formación dual (1.2.4), para ello usamos algunas propiedades de la solución óptima.

Sea $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ una cantidad denominada costo reducido de (i,j), podemos reescribir la restricción (1.2.4b) como

$$\alpha_{ij} \ge -c_{ij}^{\pi} \tag{1.2.5}$$

El coeficiente asociado con la variable α_{ij} en la función objetivo dual (1.2.4a) es $-u_{ij}$. Como deseamos maximizar el valor de la función objetivo, en cualquier solución óptima se asignara el valor más pequeño posible a α_{ij}. Entonces siguiendo esta observación y valorando (1.2.4c) y (1.2.5) implica que

$$\alpha_{ij} = \text{máx}\{0, -c_{ij}^{\pi}\} \tag{1.2.6}$$

Por tanto, si tenemos los valores óptimos para las variables duales $\pi(i)$, podemos calcular los valores óptimos para las variables α_{ij} usando (1.2.6). Entonces, este desarrollo nos permite eliminar las variables α_{ij} del problema dual y obteniendo la nueva formulación del problema dual:

$$\text{Maximizar} \quad w(\pi) = \sum_{i\in N} b(i)\pi(i) - \sum_{(i,j)\in A} \text{máx}\{0, -c_{ij}^{\pi}\}u_{ij} \tag{1.2.7}$$

Así pues el problema dual se reduce a encontrar el vector π que optimice (1.2.7).

Condiciones de Optimalidad de Holgura Complementaria

Las condiciones que vamos a enunciar permiten verificar si un flujo factible para el PFCM (1.2.3) es óptimo o no.

Teorema 1.1 (Condiciones de optimalidad de holgura complementaria, CHC)**.** *Una solución factible x^* es una solución óptima del PFCM si y solo si para un cierto conjunto de nodos potenciales π^*, los costos reducidos y los valores de flujo satisfacen las siguientes condiciones, denominadas de optimalidad de holgura complementaria para cada arco $(i,j) \in A$.*

$$Si \quad c_{ij}^{\pi^*} > 0 \qquad \Longrightarrow \quad x_{ij}^* = 0 \tag{1.2.8a}$$
$$Si \quad c_{ij}^{\pi^*} = 0 \qquad \Longrightarrow \quad 0 < x_{ij}^* < u_{ij} \tag{1.2.8b}$$
$$Si \quad c_{ij}^{\pi^*} < 0 \qquad \Longrightarrow \quad x_{ij}^* = u_{ij} \tag{1.2.8c}$$

Como podemos observar estas condiciones son las condiciones de holgura complementaria para los problemas de programación lineal cuyas variables tienen un limite superior.

Si tuviera limite inferior distinto de cero, es decir, el PFCM (1.2.1), las CHC quedarían de la siguiente manera:

$$\text{Si} \quad c_{ij}^{\pi^*} > 0 \qquad \Longrightarrow \quad x_{ij}^* = l_{ij} \tag{1.2.9a}$$
$$\text{Si} \quad c_{ij}^{\pi^*} = 0 \qquad \Longrightarrow \quad l_{ij} < x_{ij}^* < u_{ij} \tag{1.2.9b}$$
$$\text{Si} \quad c_{ij}^{\pi^*} < 0 \qquad \Longrightarrow \quad x_{ij}^* = u_{ij} \tag{1.2.9c}$$

Condiciones de Optimalidad del Costo Reducido

Teorema 1.2 (Condiciones de optimalidad de costos reducidos)**.** *Una solución factible x^* es una solución óptima del PFCM si y solo si algún conjunto de nodos potenciales π^* cumple las siguientes condiciones de optimalidad de costo reducido sobre los arcos de la red residual:*

$$c_{ij}^{\pi^*} \geq 0 \quad \textit{para todo arco} \quad (i,j) \in G(x^*) \tag{1.2.10}$$

En el teorema anterior, caracterizamos un flujo óptimo x^* como un flujo que satisfecho las condiciones (1.2.10) para algún conjunto de nodos potenciales π^*. Este conjunto se puede denominar como el conjunto de *nodos potenciales óptimos*.

Como el costo reducido es aplicable tanto a la red residual como a la red original, definimos los costos reducidos en la red residual al igual que hicimos con los costos, pero ahora usamos c_{ij}^{π} en lugar de c_{ij}.

El desarrollo del concepto de costo reducido nos lleva a las siguientes propiedades:

Propiedad 1.1
(a) Para cualquier camino dirigido P desde el nodo k al nodo l,

$$\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l) \tag{1.2.11}$$

(b) Para cualquier ciclo dirigido W,

$$\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} c_{ij} \tag{1.2.12}$$

De estas propiedades podemos obtener distintas consecuencias, la más destacable es que los potenciales de los nodos no modifican la longitud de las rutas entre cualquier par de nodos k y l, esto se debe a que los potenciales aumentan dicha longitud en una cantidad constante $\pi(k) - \pi(l)$.

Como veremos más adelante, para poder resolver PFCM utilizando el algoritmo primal-dual, empleamos c_{ij}^{π} en lugar de c_{ij}. Por tanto, es importante explicar la relación entre las funciones objetivo que describen. Antes de ello, vamos a realizar un apunte en la notación que hemos explicado para entender mejor el resultado.

Renombramos $z(x) := z_0(x) = \sum_{(i,j)\in A} c_{ij}x_{ij}$ y $z_{\pi}(x) = \sum_{(i,j)\in A} c_{ij}^{\pi}x_{ij}$.

Propiedad 1.2
Los problemas de flujo de costo mínimo con costos de arco c_{ij} o c_{ij}^{π} tienen las mismas soluciones óptimas. Además, $z_{\pi}(x) = z_0(x) - \sum_{i\in N} b(i)\pi(i) = z_0(x) - \pi b$

Capítulo 2

Algoritmo primal-dual

El algoritmo primal-dual, PD, para el PFCM emplea un pseudoflujo que satisface las condiciones de optimalidad de costo reducido (o equivalentemente las CHC) y plantea el movimiento de flujo utilizando los caminos de costo cero sobre la red con costos reducidos (en lugar de los originales).

Para cualquier pseudoflujo x, definimos el *desequilibrio* del nodo i como

$$e(i) = b(i) + \sum_{\{\, j:(j,i)\in A \,\}} x_{ji} - \sum_{\{\, j:(i,j)\in A \,\}} x_{ij} \quad \forall i \in N$$

Si $e(i) > 0$ para algún nodo i, nos referimos a $e(i)$ como el *exceso* en el nodo i. Si $e(i) < 0$ para algún nodo i, nos referimos a $-e(i)$ como el *defecto* en el nodo i. Finalmente, un nodo i está *equilibrado* si $e(i) = 0$. Además, un arco es *balanceado* si $c_{ij}^{\pi} = 0$.

El algoritmo PD transforma el problema de flujo de costo mínimo en un problema con un solo nodo *exceso* y un solo nodo con *defecto*.

Transformamos el problema introduciendo un nodo *fuente* s y un nodo *destino* t. Para cada nodo i con $b(i) > 0$, añadimos un arco (s,i) con coste cero y capacidad $b(i)$ y para cada nodo $b(i) < 0$, añadimos un arco (i,t) con coste cero y con capacidad $-b(i)$. Finalmente, obtenemos $b(s) = \sum_{\{i\in N:b(i)>0\}} b(i)$, $b(t) = -b(s)$, y $b(i) = 0 \ \ \forall i \in N$. Es fácil ver que un flujo de costo mínimo en la red transformada da un flujo de costo mínimo en la red original, ya que la red transformada se ha creado a partir de la red original unificando los valores iniciales y finales sin añadir ningún costo. Para simplificar la notación, representamos la red transformada como $G = (N,A)$, al igual que en la red original.

El algoritmo primal-dual resuelve un problema de máximo flujo en un subgrafo de la red residual $G(x)$, a la cual llamamos *red admisible*, que representamos como $G^{\circ}(x)$. Esta red se define con respecto a un pseudoflujo x que satisface las condiciones de optimalidad de costo reducido para algún potencial π, y contiene solo aquellos arcos con un costo reducido cero (balanceados) en $G(x)$. Entonces el algoritmo trabaja enviando la máxima cantidad de flujo desde el nodo con $e(i) < 0$ a nodos con $e(i) > 0$ utilizando arcos que tienen costo cero respecto a los costos reducidos definidos por ese potencial π.

La capacidad residual de un arco en $G^{\circ}(x)$ es la misma que en $G(x)$. Además, por la propia construcción de $G^{\circ}(x)$ observamos que todo camino dirigido desde el nodo s hasta el nodo t en $G^{\circ}(x)$ es también el camino más corto en $G(x)$ entre el mismo par de nodos.

Para demostrar la convergencia del algoritmo se utiliza que en cada iteración o bien se reduce el exceso y defecto de dos nodos, manteniendo el resto igual, o bien se produce un cambio de potenciales que permite etiquetar un nuevo nodo con lo que finalmente se reduce el exceso y defecto de 2 nodos.

Describimos el algoritmo PD paso a paso, trabajando directamente sobre la red original:

- *Paso 0*: Inicialización

  Definimos $\pi = 0$ y $c_{ij}^{\pi} = c_{ij}\ \forall (i,j)$ y fijamos los valores de x_{ij} de acuerdo a

  $$\begin{cases} \text{si} \quad c_{ij}^{\pi} < 0 & \Longrightarrow \quad x_{ij} = u_{ij} \\ \text{si} \quad c_{ij}^{\pi} \geq 0 & \Longrightarrow \quad x_{ij} = 0 \end{cases}$$

  Calculamos $e(i)\ \forall i$ de acuerdo a los x_{ij} establecidos.

- *Paso 1*: Construcción de I

  Seleccionamos un conjunto I de nodos i con $e(i) > 0$. Si $I \neq \emptyset$, establecemos $L := I$ y $S := \emptyset$, en otro caso, $I = \emptyset$, tendremos que el par (x, π) será óptimo si $e(i) = 0\ \forall i$ y de lo contrario el problema será no factible.

- *Paso 2*: Elegir un nodo para escanear

  Si $S = L$, vamos al *Paso* 5; si no es así seleccionamos un nodo $i \in L - S$ y establecemos $S := S \cup \{i\}$.

- *Paso 3*: Etiquetar los nodos vecinos de i

  Añadimos a L todos los nodos $j \notin L$ tal que:

  - si (i,j) es balanceado y cumple $x_{ij} < u_{ij}$, le añadimos la etiqueta (i,j) (etiquetado hacia adelante)
  - si (j,i) es balanceado y cumple $0 < x_{ji}$, le añadimos la etiqueta (j,i) (etiquetado hacia atrás).

  Si para todos los nodos j que acabamos de añadir a L tenemos $e(j) \geq 0$, vamos al *Paso* 2. De lo contrario, seleccionamos uno de estos nodos j con $e(j) < 0$ y continuamos en el *Paso* 4.

- *Paso 4*: Aumentar el flujo

  Hemos encontrado una ruta de aumento P que comienza en un nodo $i_0 \in I$ y termina en un nodo j_0 del *Paso* 3 para el que $e(j_0) < 0$. El camino se construye siguiendo las etiquetas a partir de j_0.

  $$\text{Si } j_0 \text{ tiene etiqueta} \begin{cases} (i, j_0) \text{ entonces incorporamos } (i, j_0) \text{ a } P^+ \\ (j_0, i) \text{ entonces incorporamos } (j_0, i) \text{ a } P^- \end{cases}$$

  donde P^+ y P^- son los conjuntos de arcos hacia adelante y hacia atrás de P, respectivamente. Después pasamos al nodo i y repetimos el proceso hasta llegar al nodo i_0.

  Así definimos, $\delta = min\{e(i_0), -e(j_0), \{u_{ij} - x_{ij} | (i,j) \in P^+\}, \{x_{ij} | (i,j) \in P^-\}\}$.

  Aumentamos y disminuimos en δ los flujos de todos los arcos en P^+ y P^-, respectivamente.

  Actualizamos $e(i_0)$ y $e(j_0)$, y vamos a la siguiente iteración (*Paso* 1).

- *Paso 5*: Cambiar el precio

  Sea

  $$\begin{aligned} \gamma = min\{ & \{c_{ij}^{\pi} | (i,j) \in A,\ x_{ij} < u_{ij},\ i \in S,\ j \notin S\}, \\ & \{-c_{ji}^{\pi} | (j,i) \in A,\ 0 < x_{ji},\ i \in S,\ j \notin S\}\} \end{aligned} \tag{2.0.1}$$

  Actualizamos:

  $$c_{ij}^{\pi} := \begin{cases} c_{ij}^{\pi} - \gamma & si\ i \in S,\ j \notin S\ \ \forall (i,j) \\ c_{ij}^{\pi} + \gamma & si\ i \notin S,\ j \in S\ \ \forall (i,j) \\ c_{ij}^{\pi} & otro\ caso \end{cases}$$

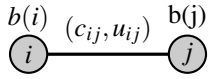
Agregamos a L todos los nodos j para los cuales el mínimo de (2.0.1) es alcanzado por un arco (i,j), y le añadimos el etiquetado hacia adelante, o es alcanzado por un arco (j,i) y le añadimos el etiquetado hacia atrás.

Si para todos los nodos j que acaban de añadirse a L, tenemos $e(j) \geq 0$, vamos al *paso* 2. De lo contrario, seleccionamos uno de estos nodos j con $e(j) < 0$ y vamos al *paso* 4.

Si no hay arco (i,j) con $x_{ij} < u_{ij},\ i \in S,\ j \notin S$, o arco (j,i) con $0 < x_{ji},\ i \in S,\ j \notin S$, el problema es no factible y el algoritmo termina.

Ahora vamos a explicar un ejemplo para ver el funcionamiento del mismo.

Primero, dado el ejemplo numérico en la parte izquierda de la figura (2.1), se añaden el nodo fuente s y el nodo destino t y los arcos $(s,1)$, $(s,2)$, $(3,t)$, $(4,t)$ con coste cero y capacidad $b(1)$, $b(2)$, $b(3)$, $b(4)$, respectivamente a cada arco. Así, se obtiene una red transformada con nuevas ofertas/demandas. Para representarlo se utiliza la siguiente notación:



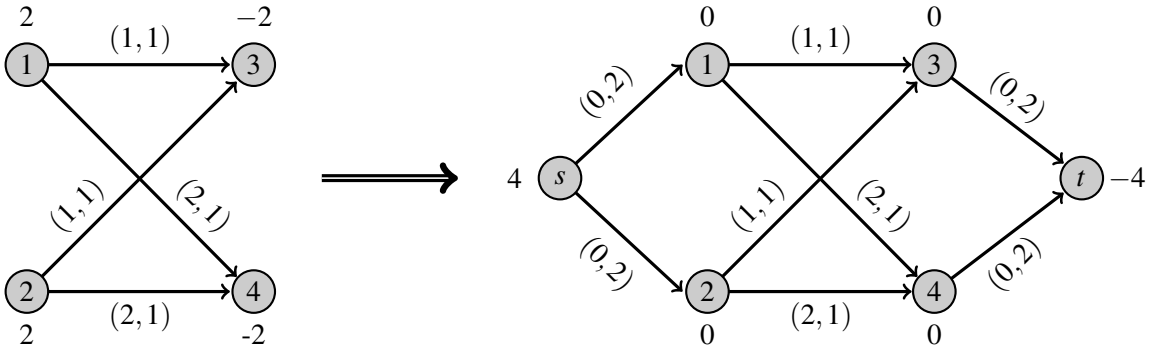
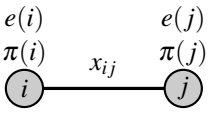
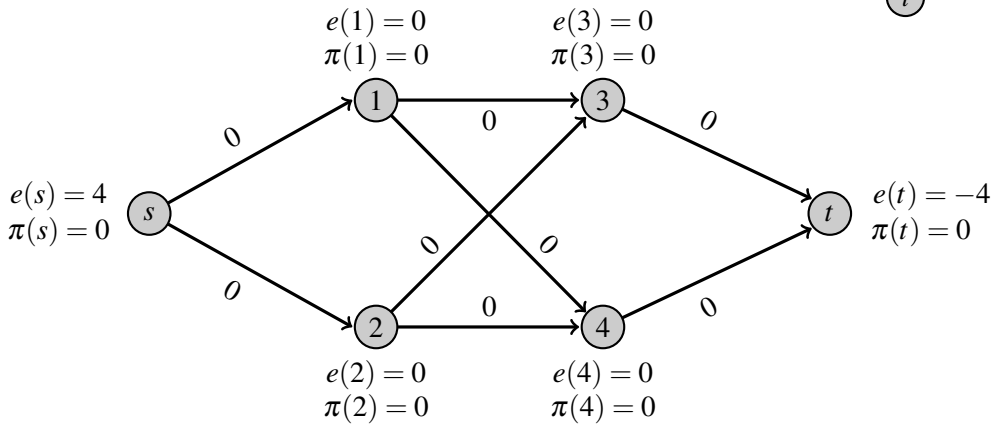


Figura 2.1: **Ejemplo Primal- Dual**

Ahora seguimos con los pasos del algoritmo. Empezamos con el *Paso 0*, definimos $\pi = 0\ \forall i \in N$, $c^{\pi}_{ij} = c_{ij}\ \forall (i,j)$ y fijamos los valores de x_{ij}. Además, calculamos los valores de $e(i)$ mediante la siguiente fórmula $e(i) = b(i) + \sum_{\{\ j:(j,i)\in A\ \}} x_{ji} - \sum_{\{\ j:(i,j)\in A\ \}} x_{ij}\ \ \forall i \in N$. Hay que tener en cuenta que se debe hacer un cambio de notación para representar la nueva red:





Una vez realizado esto, seguimos con el siguiente paso. Construimos $I = \{i | e(i) > 0\} = \{s\}$ y establecemos $L = I$ y $S = \emptyset$. Vemos que $L \neq S$, por lo que definimos $i \in L - S$, y obtenemos $i = \{s\}$ y $S = S \cup \{i\} = \{s\}$.

Continuamos con el *Paso* 3 etiquetando los nodos vecinos a s. Añadimos a L los nodos $1,2$, ya que ambos son balanceados y cumplen $x_{ij} < u_{ij}$. Como $e(j) \geq 0 \; \forall j \in L$, volvemos al *Paso* 2.

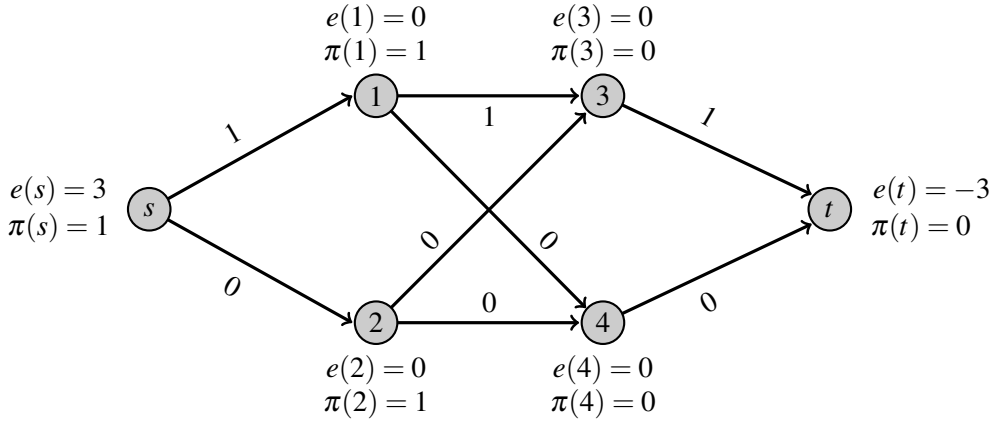
Ahora tenemos que $L \neq S$, ya que $L = \{s,1,2\}$ y $S = \{s\}$. Entonces elegimos $i = 1$ y definimos $S = \{s,1\}$. En este caso, no hay ningún nodo que cumpla las condiciones para etiquetarlo en el *Paso* 3, por lo que L queda de la misma manera y volvemos al *Paso* 2. Seleccionamos el nodo 2 y ocurre lo mismo. Entonces tenemos que $L = S = \{s,1,2\}$, y vamos al *Paso* 5, donde cambiamos los precios de los costos y los potenciales. Para ello, calculamos γ mediante los arcos $(1,3),\ (1,4),\ (2,3),\ (2,4)$, esto es

$$\gamma = min\{\ c^{\pi}_{13},\ c^{\pi}_{14},\ c^{\pi}_{23},\ c^{\pi}_{24}\} = 1.$$

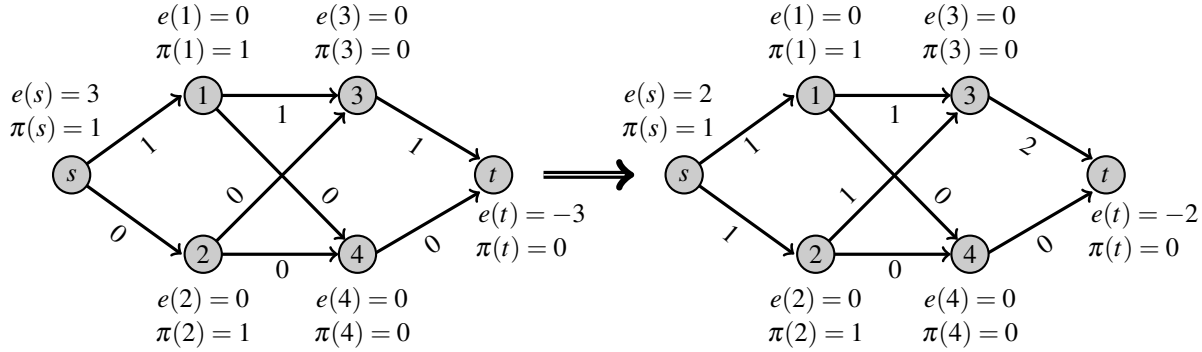
Actualizamos los costos $c^{\pi}_{13} = 0,\ c^{\pi}_{14} = 1,\ c^{\pi}_{23} = 0,\ c^{\pi}_{24} = 1$, y el resto vemos que no cambian. Agregamos a L el nodo 3, ya que cumplen el mínimo de γ, y obtenemos $L = \{s,1,2,3\}$. Como $e(j) \geq 0 \; \forall j \in L$, volvemos al *Paso* 2.

Tenemos que $L \neq S$, ya que $L = \{s,1,2,3\}$ y $S = \{s,1,2\}$. Entonces elegimos $i = 3$ y definimos $S = \{s,1,2,3\}$. Etiquetamos el arco $(3,t)$ hacia adelante y añadimos t a L, esto es, $L = \{s,1,2,3,t\}$. Como $e(t) < 0$, vamos al *Paso* 4.

Hemos encontrado una ruta de aumento P que comienza en s y termina en t mediante el camino $(s,1),\ (1,3),\ (3,t)$. Finalmente, hallamos $\delta = min\{e(s), -e(t), \{u_{s1} + x_{s1}\}, \{u_{3t} + x_{3t}\}, \{u_{13} + x_{13}\}\} = 1$ y aumentamos el flujo en dichos arcos, obteniendo el diagrama siguiente.



Se puede apreciar que cuando se aplica el algoritmo, es capaz de enviar una unidad de flujo desde el nodo s al nodo t. El algoritmo PD toma cuatro iteraciones para enviar las 4 unidades de flujo del nodo s al nodo t, en cuyo punto convierte el pseudoflujo en un flujo y termina. Representamos todas las iteraciones para ver el resultado final.



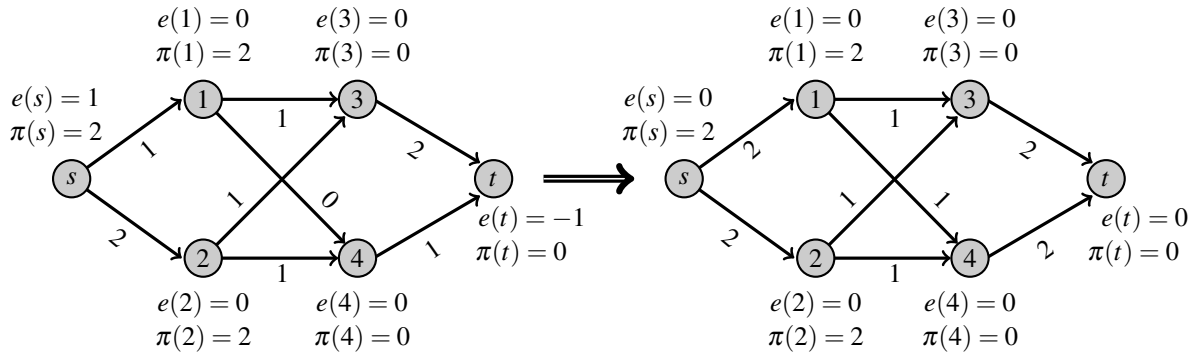


Figura 2.2: **Ejemplo Primal- Dual**

El algoritmo finaliza ya que en cada iteración, o bien se reduce el exceso de un nodo y el déficit de otro en una cantidad distinta de cero, o bien se produce un cambio de precios, pero esto se produce a lo sumo n veces ya que cada cambio permite al menos etiquetar un nodo más.

El nombre de primal-dual proviene de la teoría de la dualidad de programación lineal. Este algoritmo siempre mantiene una solución dual factible π y una solución primal que no cumple algunas restricciones de oferta/demanda, para que el par satisfaga las condiciones de holgura complementarias.

Para una solución factible dual dada, el algoritmo intenta disminuir el grado de infactibilidad primal al nivel mínimo posible. Este enfoque primal-dual es aplicable a varios problemas de optimización de combinatoria y también al problema general de programación lineal. De hecho, esta estrategia de solución dual es uno de los enfoques más populares para resolver problemas especialmente estructurados y a menudo ha producido algoritmos bastante eficientes e intuitivamente atractivos.

Capítulo 3

Algoritmo Out-of-kilter

El algoritmo *out-of-kilter*, OOK, que presentamos en esta sección, se aplica para el caso en el que el problema de flujo tiene una estructura de circulación. Para ello, hay que transformar el PFCM en un problema que no tiene excesos ni defectos en los nodos. Para lo cual, se le añade al problema de la sección anterior (2) un arco que une el nodo t y el nodo s con capacidad superior e inferior de dicho arco igual al exceso de s (o al defecto de t), dicho arco se le asigna costo c_{ts} igual a cero.

El problema queda de la siguiente forma:

$$\text{Minimizar} \quad z(x) = \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{3.0.1a}$$

sujeto a

$$\sum_{\{\, j:(i,j)\in A \,\}} x_{ij} - \sum_{\{\, j:(j,i)\in A \,\}} x_{ji} = 0 \quad \forall i \in N \tag{3.0.1b}$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \quad \forall i \in N \tag{3.0.1c}$$

El algoritmo trabaja con flujos que satisfacen las restricciones de balance de masas, pero no necesariamente las restricciones de cotas. Entonces las soluciones intermedias podrían no cumplir las condiciones de optimalidad o las restricciones de límite de flujo.

El algoritmo modifica iterativamente los flujos y potenciales de una manera que disminuye la infactibilidad de la solución y, simultáneamente, se acerca a la optimalidad.

Para describir el algoritmo, se utilizan las condiciones de optimalidad de holgura complementarias generales, CHC, indicadas en el Teorema (1.1). Para facilitar la notación, se reescriben de la siguiente forma

$$\text{Si} \quad c^{\pi}_{ij} > 0 \qquad \Longrightarrow \quad x_{ij} = l_{ij} \tag{3.0.2a}$$

$$\text{Si} \quad c^{\pi}_{ij} = 0 \qquad \Longrightarrow \quad l_{ij} < x_{ij} < u_{ij} \tag{3.0.2b}$$

$$\text{Si} \quad c^{\pi}_{ij} < 0 \qquad \Longrightarrow \quad x_{ij} = u_{ij} \tag{3.0.2c}$$

El algoritmo OOK clasifica los arcos de la red en función del cumplimiento o no de las CHC generales, es decir, los arcos están en *kilter* cuando las verifica, y *out of kilter*, OOK, cuando no las satisfacen.

El diagrama llamado *kilter*, figura 3.1, es una manera de representar dichas condiciones. La línea azul representa las zonas en las que los arcos estén en kilter.

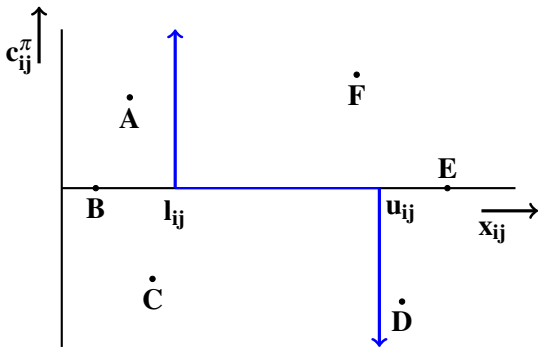


Figura 3.1: **Diagrama Kilter**

Como se puede observar, el diagrama de kilter de un arco (i, j) es la colección de todos los puntos (x_{ij}, c_{ij}^{π}) en el plano bidimensional.

La primera condición implica que si $c_{ij}^{\pi} > 0$ entonces $x_{ij} = l_{ij}$; por lo tanto, el diagrama contiene todos los puntos con x_{ij}-coordenadas l_{ij} y c_{ij}^{π}-coordenadas no negativas. De manera similar, la segunda condición produce el segmento horizontal del diagrama, y la tercera condición produce el otro segmento vertical del diagrama. Cada arco tiene su propio diagrama de kilter.

En general, dado un flujo hay que tener en cuenta que para cada arco (i, j), el flujo y el costo reducido definen un punto (x_{ij}, c_{ij}^{π}) en el plano bidimensional. Si el punto (x_{ij}, c_{ij}^{π}) se encuentra en las líneas azules del diagrama, el arco está en kilter; de lo contrario, es out-of-kilter.

Definimos el *número de kilter* k_{ij} de cada arco $(i, j) \in A$ como la magnitud del cambio en x_{ij} requerido para hacer del arco un arco en kilter manteniendo c_{ij}^{π} fijo.

Por lo tanto, de acuerdo con las condiciones primera y tercera,

$$\text{Si} \quad c_{ij}^{\pi} > 0 \qquad \Longrightarrow \quad k_{ij} = |x_{ij} - l_{ij}| \tag{3.0.3a}$$

$$\text{Si} \quad c_{ij}^{\pi} < 0 \qquad \Longrightarrow \quad k_{ij} = |u_{ij} - x_{ij}| \tag{3.0.3b}$$

$$\text{Si} \quad c_{ij}^{\pi} = 0 \quad y \quad x_{ij} > u_{ij} \quad \Longrightarrow \quad k_{ij} = x_{ij} - u_{ij} \tag{3.0.3c}$$

$$\text{Si} \quad c_{ij}^{\pi} = 0 \quad y \quad x_{ij} < l_{ij} \quad \Longrightarrow \quad k_{ij} = l_{ij} - x_{ij} \tag{3.0.3d}$$

El número de kilter de cualquier arco en kilter es cero. La suma $K = \sum_{(i,j)\in A} k_{ij}$ de todos los números de kilter nos proporciona una medida de cómo de lejos está la solución actual de la optimalidad; cuanto más pequeño es el valor de K, más cerca está la solución actual de ser una solución óptima.

Para comenzar la aplicación del algoritmo se pueden construir un flujo y un conjunto de potenciales de la forma siguiente, $\pi_j = 0 \ \forall i \in N$ y $x_{ij} = 0 \ \forall (i, j) \in A$. Posteriormente, el algoritmo mantiene todos los arcos que están en kilter y transforma sucesivamente los arcos out-of-kilter en arcos que está en kilter. El algoritmo termina cuando todos los arcos en la red se convierten arcos en kilter.

Para demostrar la convergencia del algoritmo se utiliza que los números de kilter de los arcos no aumentan. Esto lo vemos en los siguientes lemas.

Lema 3.1
La actualización de los potenciales de nodo no aumenta el número de kilter de ningún arco en la red residual.

Lema 3.2
El aumento de flujo a lo largo del ciclo dirigido $W = P \cup \{(p,q)\}$ no aumenta el número de kilter de ningún arco en la red y disminuye estrictamente el número de kilter del arco (p,q).

A continuación se describe paso a paso el algoritmo.

- *Paso 0*: Inicialización

  Definimos $\pi(i) = 0 \ \forall i \in N$, $c_{ij}^{\pi} = c_{ij}$ y $x_{ij} = 0 \ \forall (i,j) \in A$. Calculamos $k_{ij} \ \forall (i,j)$.

- *Paso 1*: Construcción de L

  Seleccionamos un arco (i_0, j_0) out-of-kilter. Si no es posible, entonces la solución actual es óptima; de lo contrario definimos, según el estado de los arcos en la figura 3.1,

  $$\begin{cases} \text{si está en la zona A, B o C} & \Longrightarrow \quad d = j_0, \ b = i_0 \\ \text{si está en la zona E, F o G} & \Longrightarrow \quad d = i_0, \ b = j_0 \ . \end{cases}$$

  Establecemos $L = \{d\}$ y $S = \emptyset$.

- *Paso 2*: Elegir un nodo para escanear

  Si $S = L$, vamos al *Paso* 5; si no es así seleccionamos un nodo $i \in L - S$ y establecemos $S := S \cup \{i\}$.

- *Paso 3*: Etiquetar los nodos vecinos de i

  Añadimos a L todos los nodos $j \notin L$ tal que (i,j) cumple

  - $c_{ij}^{\pi} > 0$ y $x_{ij} < l_{ij}$
    Le añadimos a j la etiqueta (i,j) y la capacidad residual $b_{ij} = l_{ij} - x_{ij}$, o
  - $c_{ij}^{\pi} \leq 0$ y $x_{ij} < u_{ij}$
    Le añadimos a j la etiqueta (i,j) y la capacidad residual $b_{ij} = u_{ij} - x_{ij}$.

  Añadimos a L todos los nodos $j \notin L$ tal que (j,i) cumple

  - $c_{ji}^{\pi} \geq 0$ y $x_{ji} > l_{ji}$
    Le añadimos a j la etiqueta (j,i) y la capacidad residual $b_{ji} = x_{ji} - l_{ji}$, o
  - $c_{ji}^{\pi} < 0$ y $x_{ji} > u_{ji}$
    Le añadimos a j la etiqueta (j,i) y la capacidad residual $b_{ji} = x_{ji} - u_{ji}$.

  Si para alguno de los nodos j que acabamos de añadir a L tenemos $j = b$, vamos al *Paso* 4. De lo contrario, vamos al *Paso* 2.

- *Paso 4*: Aumentar el flujo

  Hemos encontrado una ruta de aumento P que comienza en el nodo $d \in L$ y termina en el nodo b fijado en el *Paso* 1. Sea $\delta = min\{b_{i_0 j_0}, \{b_{ij} | (i,j) \in P\}$ donde

  $$b_{i_0 j_0} = \begin{cases} l_{i_0 j_0} - x_{i_0 j_0} & \text{si } (i_0, j_0) \in \text{A} \\ u_{i_0 j_0} - x_{i_0 j_0} & \text{si } (i_0, j_0) \in \text{B o C} \\ x_{i_0 j_0} - u_{i_0 j_0} & \text{si } (i_0, j_0) \in \text{D} \\ x_{i_0 j_0} - l_{i_0 j_0} & \text{si } (i_0, j_0) \in \text{E o F} \end{cases}$$

  Aumentamos y disminuimos en δ los flujos de los arcos en P^+ y P^-, respectivamente. Si (i_0, j_0) está en el A, B o C, sumar δ a $f_{i_0 j_0}$ y si (i_0, j_0) está en el D, E o F, restar δ a $f_{i_0 j_0}$.

  Finalmente, actualizamos $k_{i,j} \ \forall (i,j) \in A$ y vamos a la siguiente iteración (*Paso* 1).

- *Paso 5*: Cambiar los potenciales

  Sea

$$\begin{aligned}\gamma = min\{\{c_{ij}^{\pi} | (i,j) \in A,\ x_{ij} \leq u_{ij},\ c_{ij}^{\pi} > 0,\ i \in S,\ j \notin S\},\\ \{-c_{ji}^{\pi} | (j,i) \in A,\ l_{ji} \leq x_{ji},\ c_{ji}^{\pi} < 0,\ i \in S,\ j \notin S\}\}\end{aligned} \tag{3.0.4}$$

  Fijamos

$$c_{ij}^{\pi} := \begin{cases} c_{ij}^{\pi} - \gamma & si\ i \in S,\ j \notin S\ \ \forall (i,j) \\ c_{ij}^{\pi} + \gamma & si\ i \notin S,\ j \in S\ \ \forall (i,j) \\ c_{ij}^{\pi} & otro\ caso \end{cases}$$

  Si γ no se puede definir, el problema es no factible y el algoritmo termina.

  Si (i_0, j_0) está en orden, vamos a la siguiente iteración.

  En caso contrario, agregamos a L todos los nodos j para los cuales el mínimo de (3.0.4) es alcanzado por un arco (i,j), y le añadimos el etiquetado hacia adelante y capacidad residual $b_{ij} = u_{ij} - x_{ij}$, o es alcanzado por un arco (j,i) y le añadimos el etiquetado hacia atrás y capacidad residual $b_{ji} = x_{ij} - l_{ij}$.

  Si para todos los nodos j que acabamos de añadir a L, tenemos $j \neq b$, vamos al *paso* 2. De lo contrario, seleccionamos uno de estos nodos j con $j = b$ y vamos al *paso* 4.

Ahora vamos a explicar un ejemplo para ver el funcionamiento del mismo.

Primero, dado el mismo ejemplo numérico del problema PD (2), situado en la parte izquierda de la figura (3.2). Se añaden el nodo fuente s y el nodo destino t de la misma forma que en dicho ejemplo y además, se añade el arco (t,s) con coste cero y capacidad superior e inferior de dicho arco igual a $b(s)$. Así, se obtiene una red transformada con nuevas ofertas/demandas. Para representarlo se utiliza la siguiente notación:

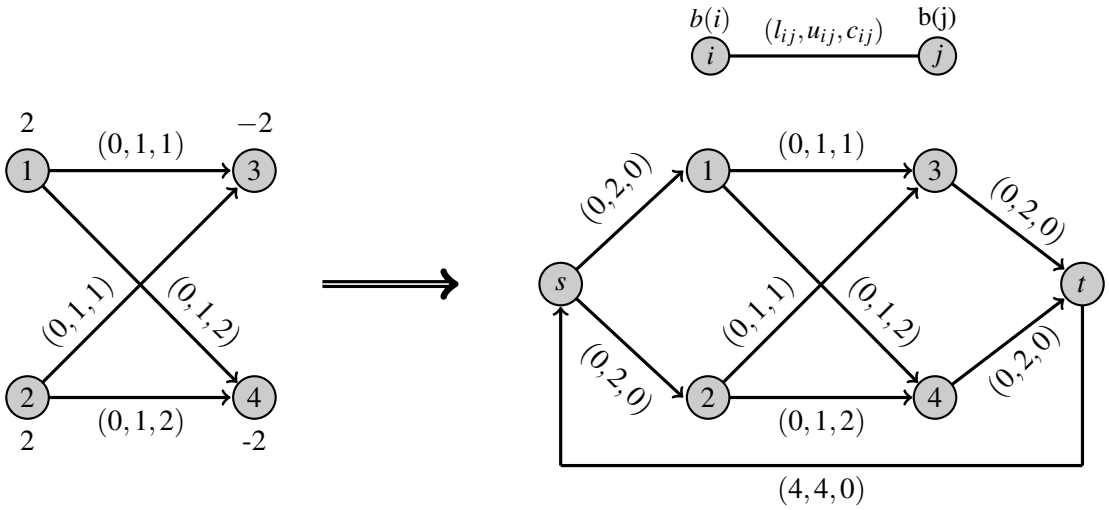
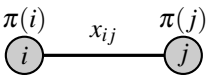
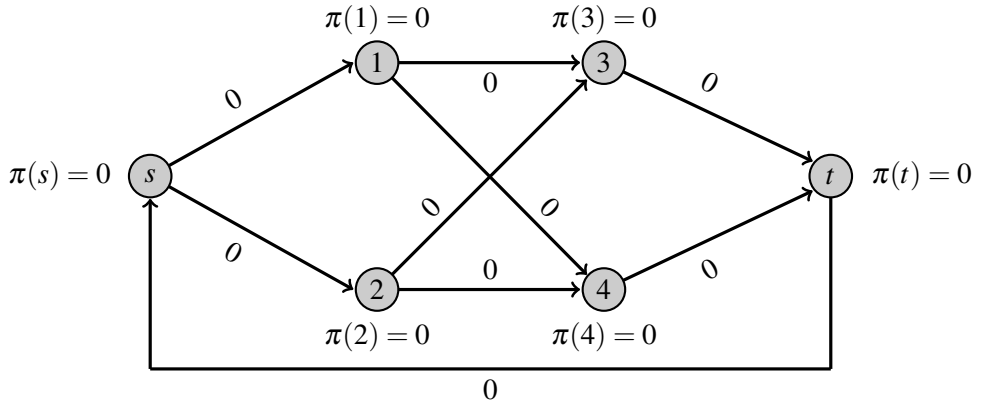


Figura 3.2: **Ejemplo Primal- Dual**

Ahora seguimos con los pasos del algoritmo. Empezamos con el *Paso 0*, definimos $\forall i \in N\ \pi(i) = 0$, y $\forall (i,j) \in A\ c_{ij}^{\pi} = c_{ij}$ y $x_{ij} = 0$. Además, calculamos los valores de $k_{ij}\ \forall (i,j) \in A$ mediante las fórmulas (3.0.3). En este caso, son todos los números de kilter 0 excepto $k_{ts} = |l_{ts} - x_{ts}| = 4$. En las redes que se muestran a continuación, indicaremos sobre los nodos el valor de los potenciales y sobre los arcos el valor actual del flujo.





Una vez realizado esto, para construir L seleccionamos un arco OOK, es decir, (t,s), y como está en B, tenemos que $d = j_0 = s$ y $b = i_0 = t$. Por tanto, establecemos $L = \{d\} = \{s\}$ y $S = \emptyset$. Vemos que $L \neq S$, por lo que seleccionamos $i \in L - S$, y obtenemos $i = \{s\}$ y $S = S \cup \{i\} = \{s\}$.

Continuamos con el *Paso* 3 etiquetando los nodos vecinos a s. Añadimos a L los nodos $1,2$, ya que ambos cumplen $c_{ij}^{\pi} \leq 0$ y $x_{ij} < u_{ij}$, y calculamos $b_{s1} = u_{s1} - x_{s1} = 2$ y $b_{s2} = u_{s2} - x_{s2} = 2$. Como $j \neq b$, siendo $j = 1,2$ y $b = t$, volvemos al *Paso* 2.

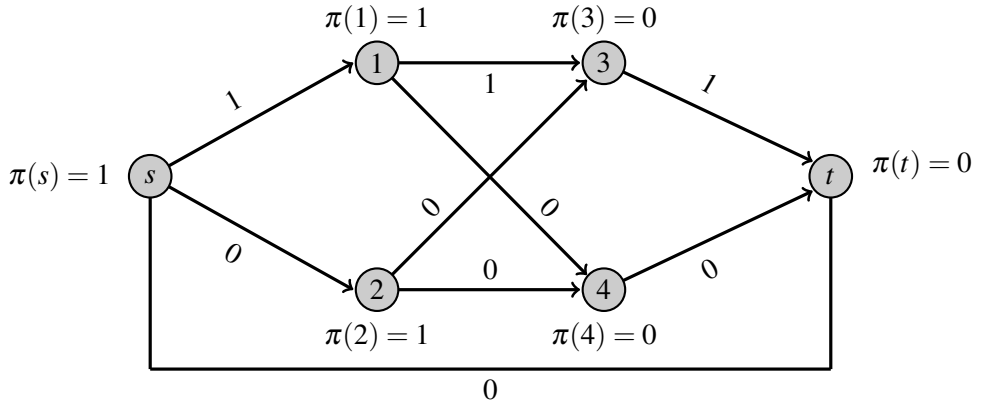
Tenemos que $L \neq S$, ya que $L = \{s,1,2\}$ y $S = \{s\}$. Entonces actualizamos $i = 1$ y definimos $S = \{s,1\}$. En este caso, no hay ningún nodo que cumpla las condiciones para etiquetarlo en el *Paso* 3, por lo que L queda de la misma manera y volvemos al *Paso* 2. Seleccionamos el nodo 2 y ocurre lo mismo. Entonces tenemos que $L = S = \{s,1,2\}$, y vamos al *Paso* 5, donde cambiamos los precios de los costos y los potenciales. Para ello, calculamos γ mediante los arcos $(1,3),\ (1,4),\ (2,3),\ (2,4)$, esto es

$$\gamma = min\{\ c_{13}^{\pi},\ c_{14}^{\pi},\ c_{23}^{\pi},\ c_{24}^{\pi}\} = 1.$$

Actualizamos los costos $c_{13}^{\pi} = 0,\ c_{14}^{\pi} = 1,\ c_{23}^{\pi} = 0,\ c_{24}^{\pi} = 1,\ c_{ts}^{\pi} = 1$, y el resto vemos que no cambian. Calculamos de nuevo $k_{ts} = |x_{ts} - l_{ts}| = 4$ y vemos que el arco (t,s) es OOK. Entonces agregamos a L el nodo 3, ya que cumplen el mínimo de γ, y calculamos $b_{13} = u_{13} - x_{13} = 1$ y $b_{23} = u_{23} - x_{23} = 1$. Obtenemos $L = \{s,1,2,3\}$ y como $j \neq b$, siendo $j = 3$ y $b = t$, volvemos al *Paso* 2.

Tenemos que $L \neq S$, ya que $L = \{s,1,2,3\}$ y $S = \{s,1,2\}$. Entonces elegimos $i = 3$ y definimos $S = \{s,1,2,3\}$. Etiquetamos el arco $(3,t)$ hacia adelante y añadimos t a L, esto es, $L = \{s,1,2,3,t\}$. Como $j = b = t$, vamos al *Paso* 4.

Hemos encontrado una ruta de aumento P que comienza en $d = s$ y termina en $b = t$ mediante el camino $(s,1),\ (1,3),\ (3,t)$. Hallamos $\delta = min\{b_{i_0,j_0}, \{b_{ij} | (i,j) \in P\}\} = min\{b_{ts}, b_{s1}, b_{13}, b_{3t}\} = 1$ y aumentamos el flujo en dichos arcos, obteniendo el diagrama siguiente.



Para finalizar, actualizamos $k_{ts} = |x_{ts} - l_{ts}| = 3$. Y vamos a la siguiente iteración (*Paso* 1).

Se puede apreciar que cuando se aplica el algoritmo, es capaz de enviar una unidad de flujo a través del ciclo. El algoritmo OOK toma cuatro iteraciones para enviar las 4 unidades de flujo, en cuyo punto convierte el pseudoflujo en un flujo y termina. Representamos todas las iteraciones para ver el resultado final.

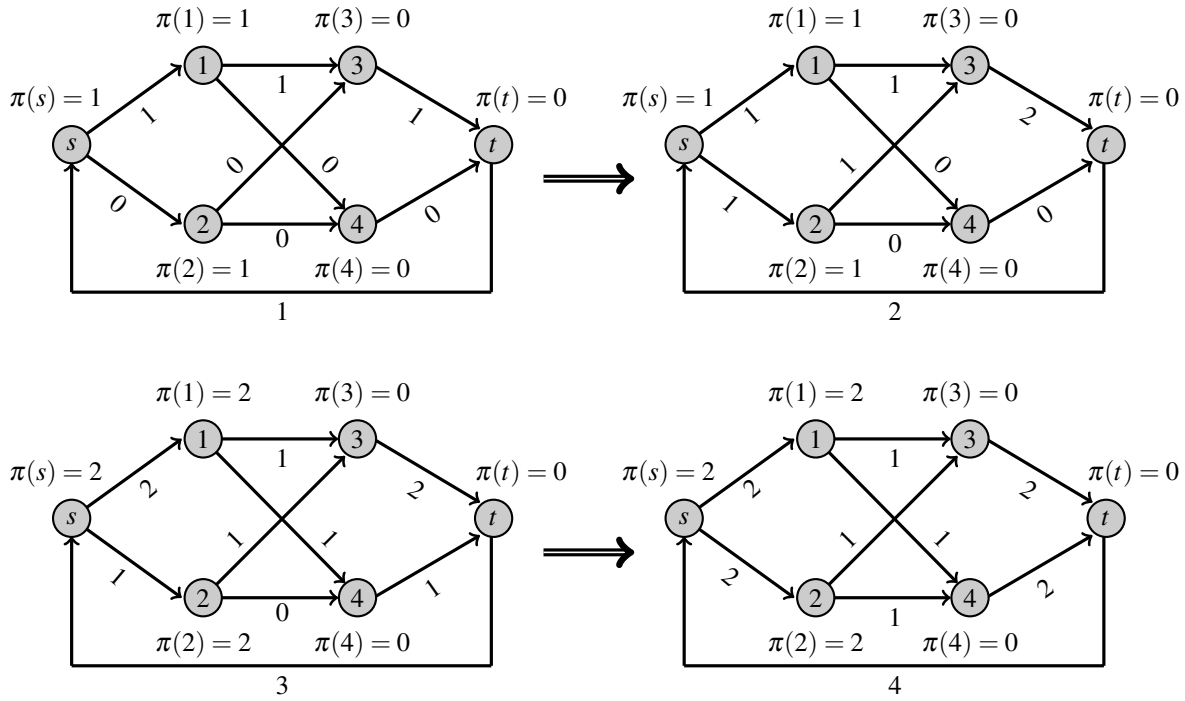


Figura 3.3: **Ejemplo Primal- Dual**

El algoritmo finaliza ya que durante las iteraciones consigue poner en kilter los arcos OOK reordenando los flujos sin forzar otro arco OOK, para ello va disminuyendo el número de kilter de dicho arco hasta llegar a cero.

Como era de esperar, hemos obtenido la misma solución que con el algoritmo PD, se envían una unidad de 1 a 3, otra de 1 a 4, otra de 2 a 3, otra de 2 a 4, con un costo de envío de 6 unidades.

Capítulo 4

Implementación y comparación

En esta sección, vamos a implementar y comparar los dos algoritmos explicados anteriormente, PD y OOK. Para ello, vamos a utilizar el lenguaje de programación *Python* [7]. Este es un lenguaje interpretado, interactivo y orientado a objetos que contiene funciones de programación avanzadas. Además, hay que tener en cuenta que el experimento se ha realizado en un ordenador Macbook Air con procesador Intel Core i5.

En el trabajo se han implementado los dos algoritmos cuyo detalle aparece en el apéndice (A) de la memoria.

Vamos a comparar los dos algoritmos para averiguar cual es mas eficiente y observar cual tiene un coste computacional menor. Para ello, hemos generado ficheros de PFCM con diferentes características mediante el generador *GRIDGEN* [8], 20 ficheros distintos de cada tipo. Inicialmente están compuestos por 10 nodos de salida y 10 nodos de llegada, y lo que variamos son los números de arcos y la oferta total. Según el número de arcos, tenemos problemas de 1000, 2000, 5000 y 10000 arcos, y según la oferta, se distinguen tres casos, problemas que ofertan 5000, 1000 y 15000 unidades de flujo.

Una vez ejecutados los problemas en cada algoritmo, se ha registrado el tiempo de ejecución en segundos y el valor del costo del envío, este último simplemente para comprobar de que ambos llegaban al mismo resultado. Posteriormente, hemos utilizado $R-Commander$ [9] para realizar los cálculos correspondientes para la comparación.

Todos estos resultados vamos a representarlos en un cuadro creado según el número de arcos y la oferta total. Podemos ver las distintas medias de tiempos de ejecución de ambos algoritmos (en segundos), y las medias del valor de la función objetivo de cada grupo de 20 problemas. Para simplificar la tabla hemos llamado al algoritmo Primal-Dual *PD*, al algoritmo Out-of-Kilter *OOK* y al valor de la función objetivo VFO.

Oferta Total	5000			10000			15000		
Arcos	PD	OOK	VFO	PD	OOK	VFO	PD	OOK	VFO
1000	4.96	4.89	673679.35	7.52	7.02	1661210.35	9.99	8.25	2748298
2000	7.69	8.7	511468.75	12.10	13.41	1384346.8	15.51	16.55	2404189.95
5000	10.98	15.31	326902.75	21.35	29.19	872191.85	31.8	41.65	1593187
10000	17.42	25.81	246852.45	32.39	49.87	645709	48.58	72.87	622799.35

Cuadro 4.1: Comparación de tiempos medios de ejecución y valor de función objetivo.

Una vez examinado el cuadro, podemos observar que el algoritmo Out-of-kilter es más rápido cuando tiene menos arcos y menor oferta total mientras que cuando ambos aumentan, se rebaja la rapidez y

el algoritmo Primal-Dual pasa a ser más rápido. En particular, en los casos de 1000 arcos, el algoritmo Primal-Dual es más lento que Out-of-kilter mientras que en los casos de 2000, 5000 y 10000 arcos, ocurre al contrario. Ahora, vamos a analizar las propiedades de los algoritmos.

- Como era de esperar, podemos ver que el tiempo de ejecución de ambos algoritmos aumenta cuando aumentan los arcos, por lo que les afecta el número de arcos que contiene cada problema.
- Además, también les afecta la cantidad de oferta total, ya que cuánta más oferta total distribuyen ambos algoritmos más tiempo tardan en obtener la solución, si bien el incremento de tiempo no es tan grande (para un mismo número de arcos).

Hemos representado los datos del cuadro en varias gráficas para ver el mejor el comportamiento de los algoritmos. Hemos diferenciado la representación de los algoritmos con distintos colores, siendo el algoritmo Primal-Dual (PD) de color amarillo y el algoritmo Out-of-kilter (OOK) de color Rojo.

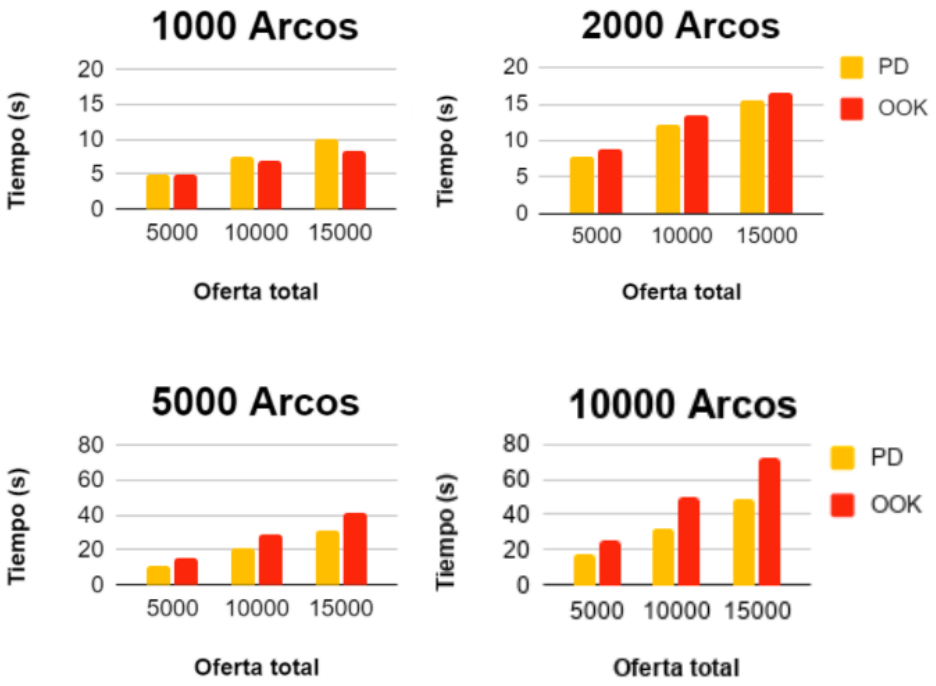


Figura 4.1: Gráficas de tiempo según la oferta total.

Se puede ver fácilmente en las gráficas anteriores lo que hemos comentado anteriormente. Ambos algoritmos aumentan en tiempo de ejecución a mayor número de arcos y mayor oferta total. Además, podemos ver que las barras correspondientes al algoritmo Primal-Dual son mayores en el caso de 1000 arcos pero en el resto de casos es menor, ya que el algoritmo Primal-Dual acelera su tiempo a más arcos, mientras que el algoritmo Out-of-kilter ralentiza su tiempo.

Para determinar si las diferencias mostradas en los resultados anteriores son significativas, calculamos un test de igualdad de medias para los tiempos, para cada uno de los grupos. Con esto conseguimos ver en que grupos se pueden considerar equivalentes los dos algoritmos y en cuales no. Este test lo realizamos mediante el Test-t de medias para datos relacionados y obtenemos que en ningún caso se pueden considerar ambos algoritmos equivalentes, (ya que en todos los casos $\alpha < 0{,}05$), así los comentarios de los párrafos anteriores son válidos. Podemos ver la tabla de los p-valores en el apéndice (B).

Para finalizar hemos realizado un cuadro para comparar las distintas desviaciones típica. Esto nos muestra la robusteza/estabilidad o no de los algoritmos. Hemos utilizado la misma notación que anteriormente para denotar los algoritmos.

Oferta Total	5000		10000		15000	
Arcos	PD	OOK	PD	OOK	PD	OOK
1000	0.69	0.61	0.86	0.70	2.56	0.92
2000	0.69	0.65	1.17	1.37	1.66	1.72
5000	1.07	1.34	1.67	2.11	2.70	3.22
10000	1.26	2.71	1.54	2.17	2.35	3.09

Cuadro 4.2: Comparación de la desviación típica.

Teniendo en cuenta el cuadro anterior, podemos observar como los datos obtenidos son considerablemente pequeños, comparados con la magnitud de los valores medios de tiempo. Los coeficientes de variación correspondientes varían de 0.04 a 0.26, lo que representan dispersiones muy pequeñas. Dichos coeficientes están desarrollados en el apéndice (B). Por tanto, la desviación típica indica que los algoritmos son estables, ya que dentro de cada grupo los tiempos de resolución de casa uno de los 20 problemas es muy similar.

Capítulo 5

Conclusión

En esta memoria, hemos estudiado el problema de flujo de costo mínimo, para lo cual hemos explicado brevemente la teoría de grafos y la dualidad.

Posteriormente, hemos explicado dos métodos habituales para resolver el PFCM, el algoritmo Primal-Dual y el algoritmo Out-of-kilter. Estos se han implementado con Python y finalmente se han comparado los resultados obtenidos al ejecutarlos.

Tras el estudio realizado, se puede concluir que el algoritmo de Primal-Dual es más eficaz que el algoritmo Out-of-kilter para casos de números grandes de arcos (considerando un número grande, un número mayor que 2000), y al contrario, en casos pequeños es más eficaz el algoritmo Out-of-kilter que el algoritmo Primal-Dual, ya que la oferta total no realiza ningún cambio destacable. Dicha oferta solo hace que ambos algoritmos demoren más el resultado.

Apéndice A

Algoritmos

En esta sección vamos a ver la implementación de los algoritmos PD y OOK. Los algoritmos se han obtenido de [2] y se han traducido a *Python* [7] desarrollando las funciones necesarias.

A.1. Algoritmo Primal-Dual

```
# IMPORTS
    import os, sys, time
    from io import open
    import numpy as np

# DATOS INICIALES
    ejercicio = open("Fichero.txt", "r")
    op = ejercicio.readlines()
    ejercicio.close()

A = []
B = []
for linea in op:
    if linea.startswith("a"):
        linea = ' '.join(linea.split())
        linea = linea.strip(' ua\n')
        lista = list(linea.split(' '))
        A.append([ int(lista[0]), int(lista[1]), int(lista[2]), int(lista[3]),
                   int(lista[4])])
    if linea.startswith("n"):
        linea = ' '.join(linea.split())
        linea = linea.strip(' un\n')
        lista = list(linea.split(' '))
        B.append([int(lista[0]), int(lista[1])])
    if linea.startswith("p"):
        linea = ' '.join(linea.split())
        linea = linea.strip(' minup\n')
        lista = list(linea.split(' '))
        N = int(lista[0])

b1 = np.zeros((N, ), dtype=int)
for j in range(N):
```

```
    for i in range(len(B)):
        if j + 1 == B[i][0]:
            b1[j] = B[i][1]
b = list(b1)
inicio_de_tiempo = time.time()

# FUNCIONES

# PASO 1: Calculamos el conjunto I y comprobamos que no es vacio
def paso1_Conjunto_I(Deseq):
    Conjunto_I = [i for i, valor in enumerate(Deseq) if valor > 0]
    if Conjunto_I == []:
        if Deseq[0] == Deseq[-1]:
            print("El par (x, pi) es optimo")
        else:
            print("El problema es no factible")
        return True
    return Conjunto_I

# PASO 2: Elegimos el nodo de escaneo
def paso2_nodo_escaneo(Conjunto_L, Conjunto_S):
    for i in Conjunto_L:
        if i not in Conjunto_S:
            Conjunto_S.append(i)
            Nodo_escaneo = i
            return Nodo_escaneo
    return "no valido"

# Funcion sumatrios de flujos
def sumatorios_flujos(Flujo, Nodos, Matriz_A):
    Sum_flujos = np.zeros((2, Nodos), dtype=int)
    # Sumatorios de flujos de salida y llegada
    for i in range(Nodos):
        for j, fila in enumerate(Matriz_A):
            if fila[0] == i:
                Sum_flujos[1][i] += Flujo[j]  # Nodos de salida
            elif fila[1] == i:
                Sum_flujos[0][i] += Flujo[j]  # Nodos de entrada
    return Sum_flujos

# Funcion desequilibrio
def desequilibrio(Vector_b, Sum_flujos, Nodos):
    Deseq = [valor_b + Sum_flujos[0][i] - Sum_flujos[1][i]
        for i, valor_b in enumerate(Vector_b) ]
    return Deseq

# PASO 3: Etiquetamos los arcos
def paso3_etiquetar(Matriz_A, Conjunto_L_S, Costo_red_Flujo, Conjunto_P,
                    Nodo_escaneo):
    Bij = 0
    for i, fila in enumerate(Matriz_A):
```

```
        if Costo_red_Flujo[0][i] == 0:
            if Costo_red_Flujo[1][i] < fila[3] and fila[0] == Nodo_escaneo:
                if fila[1] not in Conjunto_L_S[0]:
                    Bij = fila[3] - Costo_red_Flujo[1][i]
                    Conjunto_L_S[0].append(fila[1])
                    Conjunto_P.append([fila[0], fila[1], 1, Bij, i])
            elif Costo_red_Flujo[1][i] > 0 and fila[1] == Nodo_escaneo:
                if fila[0] not in Conjunto_L_S[0]:
                    Bij = Costo_red_Flujo[1][i] - fila[2]
                    Conjunto_L_S[0].append(fila[0])
                    Conjunto_P.append([fila[0], fila[1], 0, Bij, i])
    return Conjunto_L_S

# PASO 4: Encontrar ruta definitiva
def paso4_etiquetar_ruta(Conjunto_P, Nodo_inicio_ruta, Nodo_fin_ruta,
                         Conjunto_L_S):
    Conjunto_B = [[], []]
    Nodo_en_ruta = Nodo_fin_ruta
    while Nodo_en_ruta != Nodo_inicio_ruta:
        for i in range(len(Conjunto_P)):
            if Conjunto_P[i][1] == Nodo_en_ruta and Conjunto_P[i][2] == 1:
                Conjunto_B[0].insert(0, [Conjunto_P[i][0], Nodo_en_ruta,
                                         Conjunto_P[i][3], Conjunto_P[i][4] ])
                Nodo_en_ruta = Conjunto_P[i][0]
                if Nodo_en_ruta == Nodo_inicio_ruta:
                    break
            elif Conjunto_P[i][0] == Nodo_en_ruta and Conjunto_P[i][2] == 0:
                Conjunto_B[1].insert(0, [Nodo_en_ruta, Conjunto_P[i][1],
                                         Conjunto_P[i][3], Conjunto_P[i][4]])
                Nodo_en_ruta = Conjunto_P[i][1]
                if Nodo_en_ruta == Nodo_inicio_ruta:
                    break
    return Conjunto_B

# PASO 4: Calculamos la variacion del flujo
def paso4_aumento_flujo(Conjunto_B, Deseq, Costo_red_Flujo):
    Aumento_flujo = []
    for i in range(len(Conjunto_B[0])):
        Aumento_flujo.append(Conjunto_B[0][i][2])
    for i in range(len(Conjunto_B[1])):
        Aumento_flujo.append(Conjunto_B[1][i][2])
    Aumento_flujo.append(Deseq[0])
    Aumento_flujo.append(-(Deseq[-1]))
    Min_aumento_flujo = min(Aumento_flujo)
    for i, valor in enumerate(Conjunto_B[0]):
        Costo_red_Flujo[1][valor[3]] += Min_aumento_flujo
    for j, valor1 in enumerate(Conjunto_B[1]):
        Costo_red_Flujo[1][valor1[3]] += -Min_aumento_flujo
    Deseq[0] += -Min_aumento_flujo
    Deseq[-1] += Min_aumento_flujo
    return Costo_red_Flujo[1]
```

```
# PASO 5: Calculamos el precio y modificamos lo vectores
def paso5_precios(Conjunto_L_S, Matriz_A, Conjunto_P, Costo_red_Flujo):
    Precio_min = 100000
    Arco_min = list()
    Bij = 0
    for i, fila in enumerate(Matriz_A):
        if Costo_red_Flujo[0][i] != 0 and Costo_red_Flujo[1][i] < fila[3]:
            # Positivo
            if (fila[0] in Conjunto_L_S[1] and fila[1] not in Conjunto_L_S[1]):
                Precio = Costo_red_Flujo[0][i]
                if Precio < Precio_min:
                    Bij = fila[3] - Costo_red_Flujo[1][i]
                    Precio_min = Precio
                    Arco_min = list([[fila[0], fila[1], 1, Bij, i]])
                elif Precio == Precio_min:
                    Bij = fila[3] - Costo_red_Flujo[1][i]
                    Arco_min.append([fila[0], fila[1], 1, Bij, i])
        elif Costo_red_Flujo[0][i] != 0 and Costo_red_Flujo[1][i] > fila[2]:
            # Negativo
            if fila[1] in Conjunto_L_S[1] and fila[0] not in Conjunto_L_S[1]:
                Precio = -Costo_red_Flujo[0][i]
                if Precio < Precio_min:
                    Bij = Costo_red_Flujo[1][i] - fila[2]
                    Precio_min = Precio
                    Arco_min = list([[fila[0], fila[1], 0, Bij, i]])
                elif Precio == Precio_min:
                    Bij = Costo_red_Flujo[1][i] - fila[2]
                    Arco_min.append([fila[0], fila[1], 0, Bij, i])
    if Precio_min == 100000:
        print("El problema es no factible paso 5")
        return True
    if len(Arco_min) != 0:
        for ind, elemento in enumerate(Arco_min):
            Conjunto_P.append(elemento)
            if elemento[2]==1:
                Conjunto_L_S[0].append(elemento[1])
            elif elemento[2]==0:
                Conjunto_L_S[0].append(elemento[0])
    else:
        print("El problema es no factible paso 5")
        return True
    return Precio_min

# PASO 5: Modificamos los costos
def paso5_modificar(Precio_min, Costo_red_Flujo, Conjunto_L_S, Matriz_A):
    for i, fila in enumerate(Matriz_A):
        if fila[0] in Conjunto_L_S[1]:
            if fila[1] not in Conjunto_L_S[1]:
                Costo_red_Flujo[0][i] = Costo_red_Flujo[0][i] - Precio_min
        if fila[1] in Conjunto_L_S[1]:
```

```
            if fila[0] not in Conjunto_L_S[1]:
                Costo_red_Flujo[0][i] = Costo_red_Flujo[0][i] + Precio_min
    return

# ============================================
# INICIO DEL ALGORITMO

# Creamos el vector b y la matriz A incluyendo los arcos de s y t
B = []
valor_s = 0
valor_t = 0
for i1 in range(N):
    if b[i1] < 0:
        arco_nuevo_t = [i1 + 1, N + 1, 0, abs(b[i1]), 0]
        A.append(arco_nuevo_t)
        valor_t += b[i1]
        b[i1] = 0
    elif b[i1] > 0:
        arco_nuevo_s = [0, i1 + 1, 0, b[i1], 0]
        B.append(arco_nuevo_s)
        valor_s += b[i1]
        b[i1] = 0
A = B + A
b.append(valor_t)
b.insert(0, valor_s)
N = len(b)
dim_A = len(A)

# Ya hemos terminado de reconstruir la red, ahora definimos valores iniciales
costo_red_flujo = [[], []]
costo_red_flujo[0] = [fila[4] for fila in A]
costo_red_flujo[1] = [fila[3] if costo_red_flujo[0][i] < 0 else 0
                      for i, fila in enumerate(A)]
sum_flujos = sumatorios_flujos(costo_red_flujo[1], N, A)
deseq = desequilibrio(b, sum_flujos, N)  # Desequilibrio

# ITERACIONES:
FIN = False
PASO_2 = False
contador = 0
while FIN is not True:
    # Definimos los Conjuntos:
    conjunto_I = paso1_Conjunto_I(deseq)
    if isinstance(conjunto_I, bool):
        FIN = True
        break
    conjunto_L_S = [conjunto_I, []]
    conjunto_P = []
    BUSCAR_RUTA = False
    # Obtenemos una ruta viable o cambiamos los precios
    while BUSCAR_RUTA is not True:
```

```
        # Elegimos un nodo para escanearlo:
        nodo_escaneo = paso2_nodo_escaneo(conjunto_L_S[0], conjunto_L_S[1])
        # En caso de S = L vamos al paso 5
        if isinstance(nodo_escaneo, str) is True:
            precio_min = paso5_precios(conjunto_L_S, A, conjunto_P,
                                       costo_red_flujo)
            if isinstance(precio_min, bool):
                FIN = True
                break
            paso5_modificar(precio_min, costo_red_flujo, conjunto_L_S, A)
            nodo_escaneo = paso2_nodo_escaneo(conjunto_L_S[0], conjunto_L_S[1])
            # Comprobamos si es necesario dar otra vuelta al paso 2
            if deseq[conjunto_L_S[0][-1]] < 0:
                PASO_2 = True
                BUSCAR_RUTA = True
                break
        if PASO_2:
            PASO_2 = False
            continue
        paso3_etiquetar(A, conjunto_L_S, costo_red_flujo, conjunto_P,
                        nodo_escaneo)
        for j in conjunto_L_S[0]:
            if deseq[j] < 0:
                BUSCAR_RUTA = True
                break
    if FIN:
      break
    # Cuando ya hemos encontrado una ruta valida
    # Calculamos el aumento de flujo y modificamos la red
    conjunto_B = paso4_etiquetar_ruta(conjunto_P, 0, N - 1, conjunto_L_S)
    paso4_aumento_flujo(conjunto_B, deseq, costo_red_flujo)
    contador += 1

tiempo_final = time.time()
tiempo_transcurrido = tiempo_final - inicio_de_tiempo
print('Tiempo', tiempo_transcurrido)
print('Numero de iteraciones', contador)
f = 0
for i in range(len(A)):
    f += A[i][4] * costo_red_flujo[1][i]
print('VFO', f)
```

A.2. Algoritmo Out-of-kilter

```
# IMPORTS
    import os, sys, time
    from io import open
    import numpy as np

# DATOS INICIALES
```

```
    ejercicio = open("Fichero.txt", "r")
    op = ejercicio.readlines()
    ejercicio.close()

A = []
B = []
for linea in op:
    if linea.startswith("a"):
        linea = ' '.join(linea.split())
        linea = linea.strip(' ua\n')
        lista = list(linea.split(' '))
        A.append([ int(lista[0]), int(lista[1]), int(lista[2]), int(lista[3]),
                   int(lista[4])])
    if linea.startswith("n"):
        linea = ' '.join(linea.split())
        linea = linea.strip(' un\n')
        lista = list(linea.split(' '))
        B.append([int(lista[0]), int(lista[1])])
    if linea.startswith("p"):
        linea = ' '.join(linea.split())
        linea = linea.strip(' minup\n')
        lista = list(linea.split(' '))
        N = int(lista[0])

b1 = np.zeros((N, ), dtype=int)
for j in range(N):
    for i in range(len(B)):
        if j + 1 == B[i][0]:
            b1[j] = B[i][1]
b = list(b1)
inicio_de_tiempo = time.time()

# FUNCIONES

# Funcion numeros de kilter
def numeros_kilter(Costo_red_Flujo, Matriz_A):
    Dim_Matriz_A = len(Matriz_A)
    kilter = np.zeros((Dim_Matriz_A, ), dtype=int)
    for i,fila in enumerate(Matriz_A):
      if Costo_red_Flujo[0][i] > 0:
        kilter[i] = abs(Costo_red_Flujo[1][i] - fila[2])
      if Costo_red_Flujo[0][i] < 0:
        kilter[i] = abs(Costo_red_Flujo[1][i] - fila[3])
      if Costo_red_Flujo[0][i] == 0:
        if Costo_red_Flujo[1][i] > fila[3]:
          kilter[i] = Costo_red_Flujo[1][i] - fila[3]
        elif Costo_red_Flujo[1][i] < fila[2]:
          kilter[i] = -(Costo_red_Flujo[1][i] - fila[2])
    return kilter

# PASO 1: Vemos donde esta el arco y comprobamos que no es vacio
```

```
def paso1_Conjunto_R(N_kilter, Matriz_A, Costo_red_Flujo):
    Conjunto_R = []
    Bij = []
    for i, fila in enumerate(Matriz_A):
        if N_kilter[i] != 0:
            if Costo_red_Flujo[0][i] > 0:
              if Costo_red_Flujo[1][i] < fila[2]: #Arco en A
                Bij = fila[2] - Costo_red_Flujo[1][i]
                Conjunto_R= [fila[1], fila[0], 1, Bij, i]
              elif Costo_red_Flujo[1][i] > fila[2]: #Arco en F
                Bij = Costo_red_Flujo[1][i] - fila[2]
                Conjunto_R= [fila[0], fila[1], 0, Bij, i]
            if Costo_red_Flujo[0][i] == 0:
              if Costo_red_Flujo[1][i] < fila[2]: #Arco en B
                Bij = fila[3] - Costo_red_Flujo[1][i]
                Conjunto_R= [fila[1], fila[0], 1, Bij, i]
              elif Costo_red_Flujo[1][i] > fila[3]: #Arco en E
                Bij = Costo_red_Flujo[1][i] - fila[2]
                Conjunto_R[0] = [fila[0], fila[1], 0, Bij, i]
            if Costo_red_Flujo[0][i]<0:
              if Costo_red_Flujo[1][i] < fila[3]: #Arco en C
                Bij = fila[3] - Costo_red_Flujo[1][i]
                Conjunto_R= [fila[1], fila[0], 1, Bij, i]
              elif Costo_red_Flujo[1][i] > fila[3]: #Arco en D
                Bij = Costo_red_Flujo[1][i] - fila[3]
                Conjunto_R[0] = [fila[0], fila[1], 0, Bij, i]
    if Conjunto_R == []:
        print("El par (x, pi) es optimo")
        return True
    return Conjunto_R

# PASO 2: Elegimos el nodo de escaneo
def paso2_nodo_escaneo(Conjunto_L, Conjunto_S):
    for i in Conjunto_L:
        if i not in Conjunto_S:
            Conjunto_S.append(i)
            Nodo_escaneo = i
            return Nodo_escaneo
    return "no valido"

# PASO 3: Etiquetamos los arcos
def paso3_etiquetar(Matriz_A, Conjunto_L_S, Costo_red_Flujo, Conjunto_P,
                    Nodo_escaneo):
   Bij = 0
   for i, fila in enumerate(Matriz_A):
       if fila[0] == Nodo_escaneo and fila[1] not in Conjunto_L_S[0]:
          if Costo_red_Flujo[0][i] > 0 and Costo_red_Flujo[1][i] < fila[2]:
                Bij = fila[2] - Costo_red_Flujo[1][i]
                Conjunto_L_S[0].append(fila[1])
                Conjunto_P.append([fila[0], fila[1], 1, Bij, i])
          elif Costo_red_Flujo[0][i]  <= 0 and Costo_red_Flujo[1][i] < fila[3]:
```

```
                Bij = fila[3] - Costo_red_Flujo[1][i]
                Conjunto_L_S[0].append(fila[1])
                Conjunto_P.append([fila[0], fila[1], 1, Bij, i])
        elif fila[1] == Nodo_escaneo and fila[0] not in Conjunto_L_S[0]:
           if Costo_red_Flujo[0][i]  >= 0 and Costo_red_Flujo[1][i] > fila[2]:
                Bij = Costo_red_Flujo[1][i] - fila[2]
                Conjunto_L_S[0].append(fila[0])
                Conjunto_P.append([fila[0], fila[1], 0, Bij, i])
           if Costo_red_Flujo[0][i] < 0 and Costo_red_Flujo[1][i] > fila[3]:
                Bij = Costo_red_Flujo[1][i] - fila[3]
                Conjunto_L_S[0].append(fila[0])
                Conjunto_P.append([fila[0], fila[1], 0, Bij, i])
    return

# PASO 4: Encontrar ruta definitiva
def paso4_etiquetar_ruta(Conjunto_P, Nodo_inicio_ruta,
                         Nodo_fin_ruta, Conjunto_L_S):
    Conjunto_B = [[],[]]
    Nodo_en_ruta = Nodo_fin_ruta
    while Nodo_en_ruta != Nodo_inicio_ruta:
        for i in range(len(Conjunto_P)):
            if Conjunto_P[i][1] == Nodo_en_ruta and Conjunto_P[i][2]==1:
                Conjunto_B[0].insert(0, [Conjunto_P[i][0], Nodo_en_ruta,
                                         Conjunto_P[i][3], Conjunto_P[i][4]])
                Nodo_en_ruta = Conjunto_P[i][0]
                if Nodo_en_ruta == Nodo_inicio_ruta:
                  break
            elif Conjunto_P[i][0] == Nodo_en_ruta and Conjunto_P[i][2]==0:
                Conjunto_B[1].insert(0, [Nodo_en_ruta, Conjunto_P[i][1],
                                         Conjunto_P[i][3], Conjunto_P[i][4]])
                Nodo_en_ruta = Conjunto_P[i][1]
                if Nodo_en_ruta == Nodo_inicio_ruta:
                  break
    return Conjunto_B

# PASO 4: Calculamos la variacion del flujo
def paso4_aumento_flujo(Conjunto_B, Costo_red_Flujo, A, Conjunto_R):
    Aumento_flujo = []
    for i in range(len(Conjunto_B[0])):
        Aumento_flujo.append(Conjunto_B[0][i][2])
    for i in range(len(Conjunto_B[1])):
        Aumento_flujo.append(Conjunto_B[1][i][2])
    Aumento_flujo.append(Conjunto_R[3])
    Min_aumento_flujo = min(Aumento_flujo)
    for i,valor in enumerate(Conjunto_B[0]):
        Costo_red_Flujo[1][valor[3]] += Min_aumento_flujo
    for j,valor1 in enumerate(Conjunto_B[1]):
        Costo_red_Flujo[1][valor1[3]] += - Min_aumento_flujo
    if Conjunto_R[2] == 1:
      Costo_red_Flujo[1][Conjunto_R[4]] += Min_aumento_flujo
    elif conjunto_R[2] == 0:
```

```
      Costo_red_Flujo[1][Conjunto_R[4]] += - Min_aumento_flujo
    return

# PASO 5: Calculamos el precio y modificamos lo vectores
def paso5_precios(Conjunto_L_S, Matriz_A, Conjunto_P, Costo_red_Flujo):
    Precio_min = 100000
    #Arco_min = 0
    Arco_min = list()
    for i, fila in enumerate(Matriz_A):
       if Costo_red_Flujo[0][i] != 0:
          # Positivo
          if Costo_red_Flujo[1][i] <= fila[3] and Costo_red_Flujo[0][i] > 0:
             if fila[0] in Conjunto_L_S[1] and fila[1] not in Conjunto_L_S[1]:
             Precio = Costo_red_Flujo[0][i]
                if Precio < Precio_min:
                    Bij = fila[3] - Costo_red_Flujo[1][i]
                    Precio_min = Precio
                    Arco_min = list([[fila[0], fila[1], 1, Bij, i]])
                elif Precio == Precio_min:
                    Bij = fila[3] - Costo_red_Flujo[1][i]
                    Arco_min.append([fila[0], fila[1], 1, Bij, i])
          # Negativo
          elif Costo_red_Flujo[1][i] >= fila[2] and Costo_red_Flujo[0][i] < 0:
             if fila[1] in Conjunto_L_S[1] and fila[0] not in Conjunto_L_S[1]:
                Precio = - Costo_red_Flujo[0][i]
                if Precio < Precio_min:
                    Bij = Costo_red_Flujo[1][i] - fila[2]
                    Precio_min = Precio
                    Arco_min = list([[fila[0], fila[1], 0, Bij, i]])
                elif Precio == Precio_min:
                    Bij = Costo_red_Flujo[1][i] - fila[2]
                    Arco_min.append([fila[0], fila[1], 0, Bij, i])
    if Precio_min == 100000:
        print("El problema es no factible paso 51")
        return True
    if len(Arco_min) != 0:
        for ind, elemento in enumerate(Arco_min):
            Conjunto_P.append(elemento)
            if elemento[2]==1:
                Conjunto_L_S[0].append(elemento[1])
            elif elemento[2]==0:
                Conjunto_L_S[0].append(elemento[0])
    else:
        print("El problema es no factible paso 5")
        return True
    return Precio_min

# PASO 5: Modificamos los costos reducidos
def paso5_modificar_costo(Precio_min, Costo_red_Flujo, Conjunto_L_S, Matriz_A):
    for i, fila in enumerate(Matriz_A):
      if fila[0] in Conjunto_L_S[1] and fila[1] not in Conjunto_L_S[1]:
```

```
            Costo_red_Flujo[0][i] = Costo_red_Flujo[0][i] - Precio_min
      if fila[1] in Conjunto_L_S[1] and fila[0] not in Conjunto_L_S[1]:
            Costo_red_Flujo[0][i] = Costo_red_Flujo[0][i] + Precio_min
    return

# ============================================
# INICIO DEL ALGORITMO

# Creamos el vector b y la matriz A incluyendo los arcos de s y t
B = []
valor_s = 0
valor_t = 0
for i1 in range(N):
    if b[i1] < 0:
        arco_nuevo_t = [i1 + 1, N + 1, 0, abs(b[i1]), 0]
        A.append(arco_nuevo_t)
        valor_t += b[i1]
        b[i1] = 0
    elif b[i1] > 0:
        arco_nuevo_s = [0, i1 + 1, 0, b[i1], 0]
        B.append(arco_nuevo_s)
        valor_s += b[i1]
        b[i1] = 0
A = B + A
b.append(valor_t)
b.insert(0, valor_s)
arco_nuevo_ts = [N+1, 0, -b[-1], b[0], 0]
b[0] = 0
b[-1] = 0
A.append(arco_nuevo_ts)
N = len(b)
dim_A = len(A)

# Ya hemos terminado de reconstruir la red. Ahora definimos valores iniciales
costo_red_flujo = [[], []]
costo_red_flujo[0] = [fila[4] for fila in A] # Costo Reducido
costo_red_flujo[1] =  [0] * dim_A  # Flujos de arcos
n_kilter= numeros_kilter(costo_red_flujo, A) #Numeros de kilter

# ITERACIONES:
FIN = False
PASO_2 = False
contador = 0

while FIN is not True:
    # Definimos los Conjuntos:
    conjunto_R = paso1_Conjunto_R(n_kilter, A, costo_red_flujo)
    if isinstance(conjunto_R, bool):
      FIN = True
      break
    conjunto_L_S = [[conjunto_R[0]], []]
```

```
    conjunto_P = []
    BUSCAR_RUTA = False
    # Obtenemos una ruta viable o cambiamos los precios
    while BUSCAR_RUTA is not True:
        # Elegimos un nodo para escanearlo:
        nodo_escaneo = paso2_nodo_escaneo(conjunto_L_S[0], conjunto_L_S[1])
        # En caso de S = L vamos al paso 5
        if isinstance(nodo_escaneo, str) is True:
            precio_min = paso5_precios(conjunto_L_S, A, conjunto_P,
                                       costo_red_flujo)
            if isinstance(precio_min, bool):
              FIN = True
              break
            kilter = numeros_kilter(costo_red_flujo, A)
            if kilter[conjunto_R[4]] == 0:
                BUSCAR_RUTA = True
                break
            paso5_modificar_costo(precio_min, costo_red_flujo, conjunto_L_S, A)
            nodo_escaneo = paso2_nodo_escaneo(conjunto_L_S[0], conjunto_L_S[1])
            # Comprobamos si es necesario dar otra vuelta al paso 2
            for j in conjunto_L_S[0]:
                if j == conjunto_R[1]:
                  PASO_2 = True
                  BUSCAR_RUTA = True
                  break

        paso3_etiquetar(A, conjunto_L_S, costo_red_flujo,
                        conjunto_P, nodo_escaneo)
        for j in conjunto_L_S[0]:
            if j == conjunto_R[1]:
                PASO_2 = True
                BUSCAR_RUTA = True
                break
    if FIN:
      break
    conjunto_B = paso4_etiquetar_ruta(conjunto_P, conjunto_R[0],
                                      conjunto_R[1], conjunto_L_S)
    min_aumento_flujo = paso4_aumento_flujo(conjunto_B, costo_red_flujo,
                                            A, conjunto_R)
    kilter = numeros_kilter(costo_red_flujo, A)
    contador += 1

tiempo_final = time.time()
tiempo_transcurrido = tiempo_final - inicio_de_tiempo
print('Tiempo', tiempo_transcurrido)
print('Numero de iteraciones', contador)
f = 0
for i in range(len(A)):
    f += A[i][4]*costo_red_flujo[1][i]
print('VFO', f)
```

Apéndice B

Tablas necesarias

En está sección hemos representado varias tablas que hemos utilizado en el trabajo, la tabla de los p-valores del Test-t para medias y la tabla de los coeficientes de variación.

B.1. Test-t para medias

En la siguiente tabla podemos ver los resultados del Test-t para datos relacionados que hemos obtenido con R-Commander [9], está compuesta por los p-valores. Estos se calculan al comparar los algoritmos PD y OOK en cada grupo de ficheros.

Oferta Total	5000	10000	15000
1000 Arcos	0.03811	5.068e-8	0.0009139
2000 Arcos	3.205e-16	2.115e-12	1.013e-8
5000 Arcos	< 2.2e-16	< 2.2e-16	< 2.2e-16
10000 Arcos	< 2.2e-16	< 2.2e-16	< 2.2e-16

Cuadro B.1: Resultados del Test-t.

B.2. Coeficientes de variación

Para finalizar, hemos agrupado en una tabla los distintos coeficientes de variación de cada grupo de ficheros. Hemos denotado los algoritmos igual que en la sección (4) para simplificar dicha tabla.

Oferta Total	5000		10000		15000	
Arcos	PD	OOK	PD	OOK	PD	OOK
1000	0.14	0.12	0.11	0.10	0.26	0.11
2000	0.08	0.07	0.10	0.10	0.11	0.10
5000	0.10	0.09	0.08	0.07	0.09	0.07
10000	0.07	0.07	0.05	0.04	0.05	0.04

Cuadro B.2: Comparación del coeficiente de variación.

Bibliografía

[1] R. K. AHUJA, T. L. MAGNANTI Y J. B. ORLIN, *Network flows: theory, algorithms and applications*, New Jersey, 1993.

[2] D. P. BERTSEKAS, *Linear network optimization. Algorithms and codes*, MIT Press, Massachusetts, 1991.

[3] M. S. BAZARAA, J. J. JARVIS Y H. D. SHERALI, *Linear Programming and Network Flows*, New Jersey, 2010.

[4] D. P. WILLIAMSON, *Network Flows Algorithms*, Nueva York, 2019.

[5] PEDRO M. MATEO, *Optimización Lexicográfica de flujo en redes*, Tesis doctoral. Universidad de Zaragoza, 1995

[6] A. GARCÍA OLAVERRI, *, Teoría de grafos*, Universidad de Zaragoza, 2015.

[7] *Python*, `https://docs.python.org/3/whatsnew/3.9.html`.

[8] DIMACS, `http://dimacs.rutgers.edu/programs/challenge/`.

[9] *The R Project for Statistical Computing*, `https://www.r-project.org/`.