UNIVERSIDAD DE LA LAGUNA

Nuevos algoritmos y mejoras computacionales para problemas de flujos en redes

Autor: Sedeño Noda, Antonio A.

Director: Carlos González Martín

Departamento de Estadística, Investigación Operativa y Ciencias de la Computación

A mi Madre
y a Carmen

Quiero agradecer, en primer lugar, al director de esta memoria, el doctor Carlos González, por haberme introducido en este tema y por su apoyo constante en el proceso de creación y de culminación de este trabajo.

También quiero recordar los ánimos dados y las alegrías compartidas con los compañeros que han posibilitado que la labor de esta memoria haya resultado mucho más grata. Gracias a Miguel Ángel, Sergio, José Miguel, Marcos, Joaquín, Carmen Elvira y David.

Quiero dar las gracias a Carmen, por su compañía y aliento en todo momento.

Quiero expresar mi mayor agradecimiento a mi madre: ella es la razón de esta memoria.

Índice

Prólogo *i*

Capítulo 1. ***Problemas de Flujos en redes***

1. ***Introducción*** ***1***
2. ***Notación y terminología*** ***3***
   - *2.1 Representaciones de una red* *10*
   - *2.1.1 Matriz de Incidencia Nodo-Arco* *11*
   - *2.1.2 Matriz de Adyacencia Nodo-Nodo* *11*
   - *2.1.3 Listas de Adyacencia* *12*
3. ***Problemas de flujos en redes y transformaciones*** ***13***
   - *3.1 Transformaciones de la red* *17*
   - *3.1.1 Cambio de arcos no dirigidos por arcos dirigidos* *17*
   - *3.1.2 Eliminación de cotas inferiores distintas de cero* *18*
   - *3.1.3 Inversión de arcos* *19*
   - *3.1.4 Eliminación de las capacidades de los arcos* *19*
   - *3.1.5 Redes residuales* *20*
4. ***Complejidad computacional*** ***21***
   - *4.1 Diferentes medidas de la complejidad* *22*
   - *4.2 Tamaño de un problema* *24*
   - *4.3 Complejidad del caso peor* *24*
   - *4.4 Notación O grande* *25*
   - *4.5 Algoritmos polinomiales y exponenciales* *26*
   - *4.6 Notación Ω y Θ grande* *27*
   - *4.7 Recuento de operaciones representativas* *28*
5. ***Problema de flujo de coste mínimo*** ***30***
   - *5.1 Estructura básica y árboles generadores* *32*
   - *5.2 Pseudo-código del Método Simplex para redes* *34*
   - *5.3 Obtención de una estructura básica inicial* *35*
   - *5.4 Cálculo de los potenciales de los nodos y los flujos de una estructura básica* *35*
   - *5.5 Determinación del arco entrante* *37*
   - *5.6 Determinación del arco saliente* *37*
   - *5.7 Actualización de la estructura básica y de los potenciales* *38*
   - *5.8 Terminación del algoritmo* *40*
   - *5.9 Complejidad del algoritmo* *40*
6. ***Problema de flujo máximo*** ***42***
   - *6.1 Algoritmo de Ford y Fulkerson* *44*
   - *6.2 Algoritmo de caminos incrementales mínimos* *47*
   - *6.3 Escalado en las capacidades y preflujo* *52*

Capítulo 2. Algoritmos para el problema de flujo máximo

1.	***Introducción***	***59***
2.	***Estudio algorítmico del problema de flujo máximo: un análisis estadístico comparativo***	***61***
	2.1 Descripción básica de los algoritmos implementados	*63*
	2.1.1 Estrategias para mejorar el comportamiento en la práctica de los algoritmos	*67*
	2.1.2 Generadores de redes	*68*
	2.2 El experimento	*69*
	2.3 El diseño	*70*
	2.4 Análisis estadístico I	*71*
	2.5 Análisis estadístico II: Comparación de medias	*73*
	2.6 Análisis estadístico de los algoritmos individualmente	*78*
	2.7 Análisis estadístico para las variables respuesta NSAT, SAT y RET	*80*
	2.8 Resumen de conclusiones	*82*
3.	***Algoritmo de dos fases escalado en las capacidades***	***85***
	3.1 Explicación del algoritmo de dos fases escalado en las capacidades	*85*
	3.2 Un ejemplo	*87*
	3.3 Complejidad del algoritmo de dos fases escalado en las capacidades	*89*
	3.4 Mejoras del algoritmo en la práctica	*92*
	3.5 Resultados experimentales	*94*
4.	***Algoritmo de dos fases doblemente escalado en las capacidades***	***97***
	4.1 Complejidad del algoritmo de dos fases doblemente escalado en las capacidades	*100*
	4.2 Estudio computacional	*104*

Capítulo 3. Problemas de biflujo máximo

1.	***Introducción***	***113***
2.	***Preliminares y formalización del problema***	***114***
3.	***Formulación equivalente del problema BFM***	***117***
	3.1 Cambio de variables alternativo	*122*
	3.2 Resolución de P2a y P2b	*124*
4.	***Algoritmo para obtener un biflujo máximo***	***125***
	4.1 Un ejemplo	*128*
	4.2 Complejidad del algoritmo	*130*
5.	***El problema de biflujo máximo simétrico***	***131***
6.	***Algoritmo para obtener un biflujo máximo simétrico***	***133***
	6.1 Un ejemplo	*137*
	6.2 Complejidad del algoritmo	*139*
7.	***El problema de biflujo máximo biobjetivo***	***140***
8.	***Resultados computacionales***	***143***

Capítulo 4. Algoritmos para el problema de flujo de coste mínimo biobjetivo

1.	***Introducción***	***147***
2.	***Preliminares y formulación del problema***	***149***
	2.1 Caracterización de soluciones eficientes	*152*
3.	***Estudio para la distancia lineal ponderada***	***152***
	3.1 Algoritmo	*156*
	3.2 Un ejemplo	*159*
4.	***Estudio para la distancia del máximo ponderada***	***161***
	4.1 Resolución del problema P_π	*162*
	4.2 Algoritmo para el problema P_π	*165*
	4.3 Un ejemplo	*166*
	4.4 Algoritmo para el problema FRB	*169*
	4.5 Complejidad teórica del algoritmo	*172*
	4.6 Un ejemplo	*173*
5.	***Resultados computacionales de los algoritmos EEO1 y EEO2***	***174***
6.	***Caso entero***	***178***
	6.1 Obtención de todas las soluciones enteras que están sobre la frontera eficiente	*178*
	6.2 Obtención de los puntos enteros eficientes que no pertenecen a la frontera eficiente	*182*
	6.3 Algoritmo	*189*
	6.4 Complejidad teórica del algoritmo	*191*
	6.5 Un ejemplo	*192*
	6.6 Resultados computacionales	*194*

Bibliografía

Prólogo

Desde una perspectiva global, el concepto de red es de importancia relevante en la concepción y el desenvolvimiento de multitud de aspectos vitales. La idea de red aparece en procesos naturales y trasciende a fenómenos de índole organizativo y económico, sustentando estructuras de relevancia decisiva en las formas modernas de vivir.

Es habitual que por las redes circulen flujos. Una red suele ser el soporte, físico o abstracto, por el que circula uno o varios bienes que, de forma general, son ofertados y demandados por algunos puntos localizados en la red. Las conexiones en la misma, bajo factibilidad del problema, aseguran la satisfacción de los requerimientos establecidos.

El título flujos en redes constituye un campo de trabajo científico, inmerso en la Investigación Operativa, que reúne a estudiantes, practicantes e investigadores alrededor de una disciplina de contenido intelectual con un amplio rango de aplicabilidad. Como disciplina científica aparece integrada en áreas como las Matemáticas Aplicadas, las Ciencias de la Computación, Ingenierías de las Comunicaciones, Ciencias de la Gestión, Ciencias Económicas, Ciencias Experimentales, etc.. La literatura existente recorre miles de aplicaciones en campos como Química, Física, redes de computadores, Economía y Ciencias de la Gestión, Ingenierías de Telecomunicaciones y muchas otras ramas de la ingenierías, política y sistemas sociales, planificación, sanidad, etc.

Los problemas de flujos en redes, a pesar de tener antecedentes históricos como el problema de los puentes de Königsberg, tratado por Euler en el siglo XVIII, o problemas de flujos eléctricos, analizados por Gustav Kirchhoff en el siglo XIX,

son estudiados como tales a partir de la década de los años cincuenta en el marco de la Investigación Operativa.

En conexión con el desarrollo de esta rama del saber, usando descubrimientos y avances propios del contexto donde aparecen y, sobre todo, en sintonía con los adelantos en la construcción y desarrollo de nuevos computadores electrónicos, se ha intensificado de manera creciente el estudio de problemas de análisis de redes, aportándose algoritmos cada vez más eficientes desde el punto de vista práctico y computacional. Este estudio abre, de manera continuada, nuevas perspectivas para el tratamiento y resolución de problemas más complejos.

En este trabajo presentamos nuevos algoritmos para algunos problemas de flujos en redes, haciendo énfasis en los aspectos algorítmicos y proponiendo mejoras computacionales, estos últimos en sus vertientes teóricas y prácticas.

En el capítulo 1, comenzamos con una introducción que muestra la importancia de los problemas de flujos en redes. En este capítulo, introducimos los conceptos de grafos y redes necesarios para el desarrollo de la memoria. A continuación, introducimos las diferentes medidas para estimar el comportamiento computacional teórico de un algoritmo (cabe destacar entre ellas la notación O grande) y para estimar el comportamiento práctico del mismo, como son el recuento de las operaciones representativas de un algoritmo que permiten estimar el denominado tiempo de CPU virtual asociado. Seguidamente, introducimos y formalizamos dos problemas esenciales de optimización en redes: el problema de flujo de coste mínimo y el problema de flujo máximo. Para el primero describimos en detalle el Método Simplex para Redes, el cual será utilizado en el capítulo 4 de esta memoria. Para el segundo, describimos los algoritmos genéricos que sirven de base a los distintos tipos de algoritmos que se introducirán en los capítulos 2 y 3.

En el capítulo 2, centramos el estudio en el problema de flujo máximo. La primera parte de este capítulo está dedicada a estudiar el comportamiento práctico de un numeroso grupo de algoritmos. Para ello, llevamos a cabo un análisis estadístico comparativo que nos permitirá responder a una serie de preguntas previas al diseño del experimento. Este estudio es concluyente con respecto a los parámetros que influyen en el comportamiento individual de un

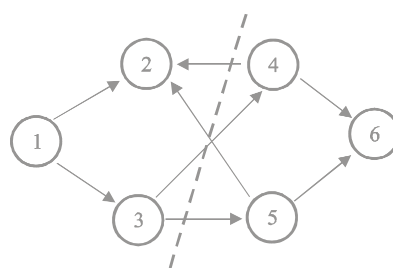
algoritmo, pero además, permite obtener conclusiones del comportamiento de grupos de algoritmos que tienen o usan herramientas comunes. Mucho de lo aprendido en el anterior estudio, nos permite, a continuación, introducir un nuevo algoritmo para el problema cuya complejidad del caso peor, para redes de capacidades “pequeñas”, es seguramente la de menor cota. Finalmente, en este capítulo generalizamos el anterior algoritmo y realizamos un estudio computacional del mismo obteniendo el tiempo de CPU virtual, o lo que es lo mismo, la función de complejidad en la práctica.

En el capítulo 3 nos dedicamos al problema de biflujo máximo. Este problema constituye para nosotros un primer acercamiento a los problemas de flujos múltiples. La resolución de este problema en su caso no dirigido es consecuencia del teorema de biflujo-máximo corte-mínimo. En la literatura disponible, las demostraciones de este teorema no son nada claras ya que la mayoría utilizan resultados no específicos de la optimización en redes. Por el contrario, en este capítulo demostramos de manera alternativa este teorema, utilizando la nomenclatura y herramientas propias de los problemas de flujos en redes. Esta demostración permite, de manera directa, obtener un algoritmo actualizado que resulta eficiente desde un punto de vista computacional. En adición, el esfuerzo realizado en el anterior problema nos permite estudiar y resolver el problema de biflujo máximo simétrico. Finalmente, en este capítulo caracterizamos las soluciones del problema de biflujo máximo biobjetivo, el cual es una generalización natural del problema de biflujo máximo. Mostramos que las soluciones eficientes de este último problema consisten en las soluciones alternativas óptimas del problema de biflujo máximo.

En el capítulo 4 nos introducimos en el campo de la optimización multiobjetivo, considerando el problema de flujo de coste mínimo biobjetivo. Para este problema distinguimos los casos en las que las variables de flujos no están restringidas a tomar valores enteros y en los que si lo están. Para el primer caso proponemos dos algoritmos distintos que usan métricas diferentes para caracterizar el conjunto eficiente en el espacio objetivo. Según la información disponible, uno de ellos resulta ser, en la práctica, el más rápido de los algoritmos referenciados en la literatura existente, al aprovechar la menor dimensión del espacio objetivo

frente a la dimensión del espacio de decisiones. Realizamos un estudio computacional que pone de manifiesto lo comentado. Finalmente, damos un algoritmo para el caso entero. Este procedimiento es uno de los pocos métodos existentes (si no es el único) para resolver globalmente el problema. En base a este nuevo algoritmo, realizamos un experimento computacional en el que se observa que el número de soluciones eficientes que no están sobre la frontera eficiente es muy superior al conjunto de estas que pertenecen a dicha frontera.

El trabajo se completa con una bibliografía que referencia los trabajos que han servido de base para confeccionar la presente memoria.



Capítulo 1

Problemas de Flujos en Redes

1. Introducción

El concepto de grafo resulta de la abstracción de situaciones reales en las que aparecen ciertos lugares o puntos con conexiones entre ellos. Por tanto, un grafo queda definido por un conjunto de vértices o nodos y un conjunto de pares de esos vértices. Cuando los elementos de un grafo tienen asociados valores numéricos o magnitudes (pesos, distancias, costos, capacidades, disponibilidades, ofertas, demandas, etc.), aparece el concepto de red. Ejemplos reales de redes resultan cotidianos y son de una importancia capital en el mundo actual: redes de comunicaciones de todo tipo (terrestres, aéreas, telefónicas,...), de distribución de bienes (eléctricas, de aguas, de gas,...), de organización y gestión (servicios, sanidad, seguridad,...), etc.. Existen muchas redes por las que circula algún tipo de flujo y, por ello, muchos de los problemas de optimización inherentes son problemas de flujos en redes.

Dado que esta memoria está dedicada al estudio de problemas de flujos en redes, en el presente capítulo haremos una introducción y un estudio básico de dichos problemas. En principio, nos centraremos en uno de los casos más generales conocido como problema de flujo de coste mínimo y en el caso particular del anterior denominado problema de flujo máximo.

El problema de flujo de coste mínimo fue introducido a principios de la década entre 1950 y 1960. En 1957, Ford y Fulkerson [29], desarrollan un algoritmo primal-dual para el problema de transporte capacitado y más tarde, en 1962, generalizan estas ideas para resolver el problema de flujo de coste mínimo. Jewell [46], Iri [44] y Busaker y Gowen [15] desarrollan, independientemente, el algoritmo de caminos mínimos sucesivos. Estos investigadores resuelven el problema de flujo de coste mínimo utilizando una secuencia de caminos mínimos con longitudes arbitrarias en los arcos. Aproximadamente, al mismo tiempo y de forma independientemente, Minty [58] y Fulkerson [30] desarrollan el algoritmo Out-of-Kilter. Un poco más tarde, Klein [50] introduce el algoritmo de cancelación de ciclos. Por otra parte,

Bertsekas y Tseng [11] desarrollan en 1988 el algoritmo de relajación, el cual es, junto con el Método Simplex para redes, uno de los algoritmos más rápidos en la práctica para resolver el problema de flujo de coste mínimo.

En 1951, Dantzig [23] desarrolla el Método Simplex para redes para el problema de transporte sin capacidades mediante una especialización de su Método Simplex. La forma actual del Método Simplex para redes es debida a las contribuciones de numerosos autores. Entre estos podemos destacar los trabajos de Johnson [48], Bazaraa, Jarvis y Sherali [10], Glover, Karney y Klingman [36], Mulvey [59], Bradley, Brown y Graves [13], Grigoriadis [42] y Chang y Chen [17].

En este capítulo describiremos en detalle el Método Simplex para redes, debido a que lo usamos como herramienta para los algoritmos desarrollados en el capítulo 4. Hemos elegido el Método Simplex para redes por dos razones. La primera es que éste resuelve cualquier problema de flujos mientras que, por ejemplo, el método Out-of-Kilter resuelve únicamente el problema de circulación. La segunda razón es que estudios computacionales muestran que el Método Simplex para redes es substancialmente más rápido que los métodos Primal-Dual y Out-of-Kilter (ver Glover et al. [36]).

Por su parte, el problema de flujo máximo es uno de los problemas de optimización en redes más extensamente estudiado y tiene numerosas aplicaciones (ver, por ejemplo, Ahuja, Magnanti y Orlin [3]). Este problema fue introducido por Fulkerson y Dantzig en 1955 y resuelto por vez primera por Ford y Fulkerson [28] con su conocido algoritmo de *caminos incrementales*. Posteriormente Dinic [25] en 1970 introduce el concepto de redes de caminos mínimos, llamadas redes estratificadas. En 1972, Edmonds y Karp [26] sugieren dos implementaciones polinomiales del algoritmo de Ford y Fulkerson. La primera envía flujo a lo largo de caminos de mayor capacidad residual, la segunda envía flujo a lo largo de caminos mínimos. Hasta este momento, todos los algoritmos de flujo máximo son algoritmos de caminos incrementales. En 1974, Karzanov [49] introduce el concepto de *preflujo* que utiliza sobre redes estratificadas. A partir de entonces, numerosos autores han diseñado algoritmos que, incorporando nuevas e importantes ideas,

resuelven el problema intentando mejorar la complejidad computacional asociada.

Los métodos más importantes son debidos a Dinic [25], Edmonds y Karp [26], Karzanov [49], Malhotra, Kumar y Maheshwari [56], Gabow [31], Sleator y Tarjan [73], Goldberg[38], Goldberg y Tarjan[37], Cheriyan y Maheshwari [19], Ahuja y Orlin [1], [5], [6] y Cheriyan, Hagerup y Melhorn [18]. Entre los estudios globales de este problema podemos destacar los debidos a Nicoloso y Simeone [61], Fernandez-Baca y Martel [27] y el excelente texto de Ahuja, Magnanti y Orlin [4]. Existen diversos estudios computacionales de los algoritmos de flujo máximo entre los que destacamos los debidos a Derigs y Meier [24] y a Ahuja et al. [2].

El trabajo que pretendemos desarrollar en este capítulo debe servir para sentar las bases del estudio de los problemas de flujos en redes. En la segunda sección relacionaremos una serie de términos básicos sobre grafos y redes. En la tercera sección abordaremos la notación habitual de los problemas de flujos en redes, así como las transformaciones posibles en dichas redes. En la cuarta sección, daremos las nociones necesarias sobre las medidas utilizadas para estimar la bondad de un algoritmo. En la quinta sección, nos dedicamos al problema de flujo de coste mínimo, exponiendo de manera exhaustiva el Método Simplex para redes. Concluimos este capítulo con la sexta sección, refiriéndonos al problema de flujo máximo donde introducimos una serie de algoritmos básicos necesarios para el desarrollo de esta memoria.

2. Notación y terminología

En los grafos y las redes se distinguen dos clases de elementos: los nodos, vértices o puntos y las conexiones, ramas o uniones. De las propiedades o características asociadas a dichos elementos derivarán una serie de conceptos cuya relación resumida aparece a continuación:

Grafos dirigidos: Dado un grafo $G=(V,A)$, donde V es el conjunto de nodos, si los elementos de A son pares ordenados de nodos, diremos que G es un grafo dirigido y esos elementos serán

denominados arcos. La Figura 1.1 da un ejemplo de grafo dirigido. Para este grafo V={1,2,3,4,5,6} y A={(1,2), (1,3), (2,3), (2,4), (3,6), (4,5), (4,6), (5,2), (5,3), (5,6)}. En general, denotaremos por n al número de nodos y por m al número de arcos de G.

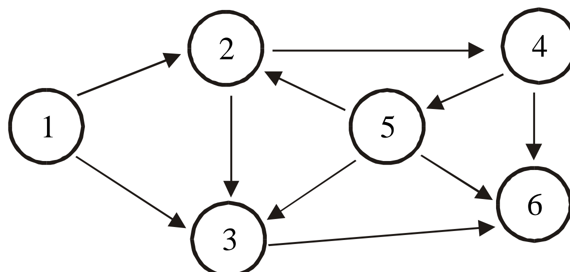


Figura 1.1. Grafo dirigido

Grafos no dirigidos: Cuando en un grafo el conjunto de conexiones está formado por pares no ordenados de nodos, tendremos un grafo no dirigido. Esos pares no ordenados se denominarán aristas o arcos no dirigidos del grafo (en este caso, las conexiones de los grafos dirigidos se entenderán que son arcos dirigidos). La Figura 1.2 da un ejemplo de grafo no dirigido.

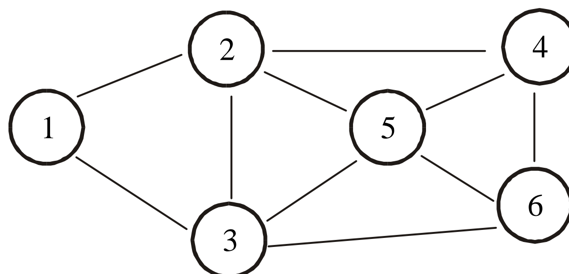


Figura 1.2. Grafo no dirigido

Por lo dicho anteriormente, se entiende fácilmente que, por extensión de las definiciones introducidas, tiene sentido referirse a redes dirigidas y no dirigidas. En el resto de esta memoria, asumiremos que las redes (en su caso, grafos) que manejamos son dirigidas, a no ser que explícitamente se indique lo contrario.

Cola y cabeza de un arco: Un arco dirigido (i,j) tiene dos puntos extremos, i y j: i será la cola del arco (i,j) y j su cabeza. Diremos que el arco (i,j) sale o emana del nodo i y termina o llega en el nodo j. También, dado (i,j), decimos que los nodos i y j son adyacentes.

Grado de un nodo: el grado de entrada de un nodo i, δ_i^-, coincide con el número de arcos que llegan a este nodo y su grado de salida, δ_i^+, coincide con el número de arcos que salen de dicho nodo. El grado de un nodo i, δ_i, coincide con la suma del grado de entrada y de salida de dicho nodo. Por ejemplo, en la Figura 1.1, para el nodo 5 se tiene que $\delta_5^- = 1$, $\delta_5^+ = 3$ y $\delta_5 = 4$. Es fácil ver que la suma de todos los grados de entrada de todos los nodos es igual a la suma de los grados de salida de todos los nodos y que ambas sumas coinciden con el número de arcos en la red.

Listas de Adyacencia: La lista de arcos adyacentes $A(i)$ de un nodo i es el conjunto de arcos que salen de este nodo, es decir, $A(i) = \{(i,j) \in A : j \in V\}$. Dado cualquier nodo i, definimos su lista de nodos predecesores y nodos sucesores mediante los conjuntos $\Pr ed(i) = \{j \in V / (j,i) \in A\}$ y $Suc(i) = \{j \in V / (i,j) \in A\}$, respectivamente. Asumiremos, sin pérdida de generalidad, que los arcos de la lista de adyacencia $A(i)$ aparecen en orden creciente de los nodos cabeza de los arcos. Debemos notar que $|A(i)| = \delta_i^+$ y, por tanto, $\sum_{i \in V} |A(i)| = m$.

Arcos múltiples y lazos: Los arcos múltiples son dos o más arcos con los mismos nodos colas y cabezas. Un lazo es un arco cuyo nodo cola coincide con su nodo cabeza. Asumiremos, salvo comentario explícito, que los grafos que utilizamos no contienen ni arcos múltiples ni lazos.

Subgrafo: Un grafo $G' = (V', A')$ es un subgrafo de $G=(V,A)$ si $V' \subseteq V$ y $A' \subseteq A$. Diremos que $G' = (V', A')$ es el subgrafo de G inducido por V' si A' contiene cada arco de A con ambos extremos en V'. Un grafo $G' = (V', A')$ es un subgrafo generador de $G=(V,A)$ si $V' = V$ y $A' \subseteq A$.

Cadena: Una cadena, en un grafo dirigido $G=(V,A)$, es un subgrafo de G que consiste de una secuencia de nodos y arcos i_1, a_1, i_2, a_2,…,i_{r-1}, a_{r-1}, i_r satisfaciendo que $a_k = (i_k, i_{k+1}) \in A$ ó $a_k = (i_{k+1}, i_k) \in A$ para todo $1 \le k \le r-1$ y ninguno de los nodos aparece repetido, es decir, $i_k \ne i_{k+1}$ con $1 \le k \le r-1$. La Figura 1.3a) ilustra el concepto de cadena en un grafo: 1-2-3-6. Podemos dividir los arcos de una cadena en dos grupos: arcos hacia delante y arcos hacia atrás. Un

arco (i, j) de la cadena es hacia delante si la cadena visita al nodo i primero que al nodo j, y en otro caso, es un arco hacia atrás. Por ejemplo, en la Figura 1.3a), los arcos (1,2) y (3,6) son hacia delante y el arco (3,2) es hacia atrás.

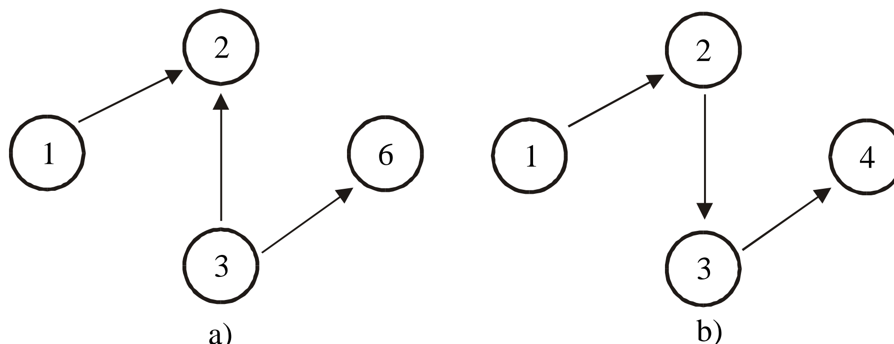


Figura 1.3. Cadenas y caminos.

Camino: Un camino es una versión orientada de una cadena en el sentido que para cualquiera dos nodos consecutivos i_k y i_{k+1} en el camino se tiene que $a_k = (i_k, i_{k+1}) \in A$. En otras palabras, un camino es una cadena sin arcos hacia detrás. La Figura 1.3b) da un ejemplo de camino: 1-2-3-4.

Ciclo: Un ciclo es una cadena i_1, $i_2,\ldots,i_{r-1}$, i_r junto con el arco (i_r, i_1) o (i_1, i_r). Dicho de otro modo, un ciclo es una cadena cerrada. La Figura 1.4a) ilustra la idea de ciclo:3-7-5-3. En un ciclo también se puede hablar de arcos hacia delante y arcos hacia atrás, una vez definida la orientación del ciclo. En la Figura 1.4a), los arcos (3,5) y (7,3) son arcos hacia atrás y el arco (7,5) es un arco hacia delante.

Circuito: un circuito es un camino i_1, $i_2,\ldots,i_{r-1}$, i_r al que se añade el arco (i_r, i_1). Es decir, un circuito es un camino cerrado. La 1.4b) muestra el circuito 1-2-5-1.

Grafo sin ciclos y sin circuitos: Un grafo es acíclico si no contiene ciclos. Un grafo se dice sin circuitos si no contiene circuitos. Un grafo acíclico es también sin circuitos pero no al revés.

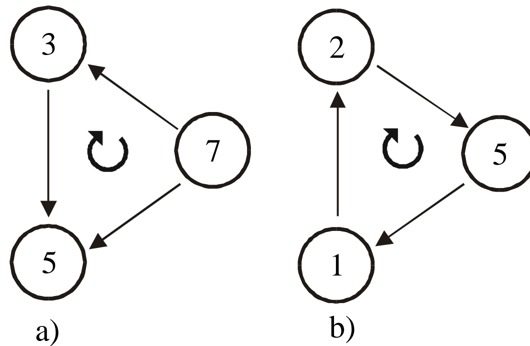


Figura 1.4. Ciclos y circuitos.

Conectividad: Decimos que dos nodos i y j están conectados si el grafo contiene al menos una cadena desde el nodo i al nodo j. Un grafo es conexo si todo par de nodos están conectados; en otro caso el grafo es no conexo. Llamaremos componentes conexas de un grafo no conexo a los subgrafos conexos de dicho grafo. Por ejemplo, el grafo mostrado en la Figura 1.5a) es conexo y el de la Figura 1.5b) es no conexo. Este último tiene 2 componentes conexas que consisten de los conjuntos de nodos {1,2,3} y{4}.

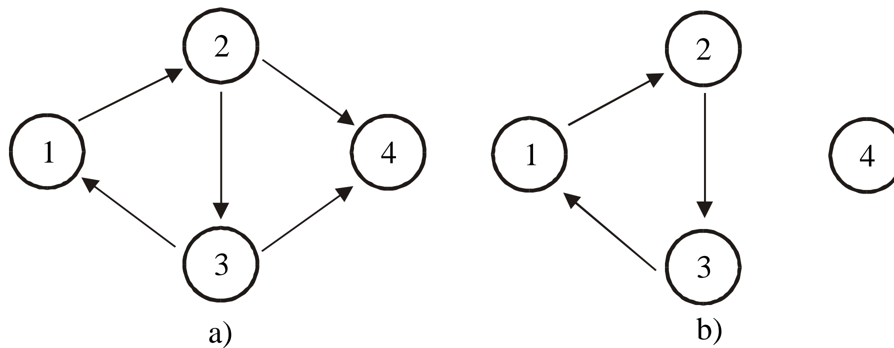


Figura 1.5. Componentes conexas.

Conectividad fuerte: Un grafo conexo es fuertemente conexo si existe al menos un camino entre cualquier par de nodos del grafo. En la Figura 1.5a) la componente definida por el conjunto de nodos {1,2,3} es fuertemente conexa, sin embargo el grafo no es fuertemente conexo debido a que no contiene caminos desde el nodo 4 al resto de los nodos.

Corte: Un corte es una partición del conjunto de nodos V en dos subconjuntos S y $\overline{S} = V - S$. Cada corte define un conjunto de arcos con un extremo en S y otro extremo en $\overline{S}$. Así, nos referiremos a este conjunto de arcos como un corte y lo representaremos mediante la notación $[S,\overline{S}]$. La Figura 1.6 ilustra

un corte con S={1,2,3} y $\overline{S}$={4,5,6}. El conjunto de arcos del corte es $[S,\overline{S}]$={(3,4),(3,5),(4,2),(5,2)}.

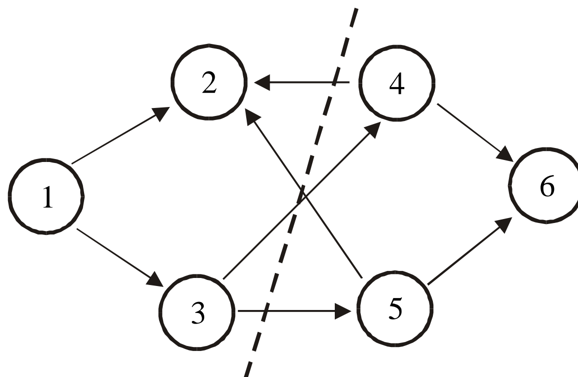


Figura 1.6. Corte.

s-t corte: Un s-t corte se define con respecto a dos nodos distinguidos s y t, y es un corte $[S,\overline{S}]$ satisfaciendo que $s \in S$ y $t \in \overline{S}$. Para el ejemplo anterior, si s=1 y t=6, el corte representado en la Figura 1.6 es un s-t corte.

Árbol: un árbol es un grafo conexo que no contiene ciclos. El concepto de árbol aparece en una gran variedad de algoritmos de flujos en redes. Asumiremos, las siguientes propiedades elementales de los árboles:

a) Un árbol de n nodos contiene exactamente n-1 arcos.

b) Un árbol tiene al menos dos nodos hoja (nodos con grado 1)

c) Cada dos nodos de un árbol están conectados por una única cadena o camino.

En la Figura 1.7 se dan ejemplos de árboles.

Bosque: Un grafo que no contiene ciclos es un bosque. Dicho de otra manera, un bosque es una colección de árboles. El grafo representado por los dos subgrafos de la Figura 1.7 es un bosque.

Subárbol: Un subgrafo conexo de un árbol es un subárbol.

Árbol enraizado: Un árbol enraizado es un árbol con un nodo especial llamado raíz. Es ilustrativo pensar en un árbol enraizado como uno que cuelga del nodo raíz. La Figura 1.8 muestra un ejemplo de árbol enraizado, la raíz es el nodo 1.

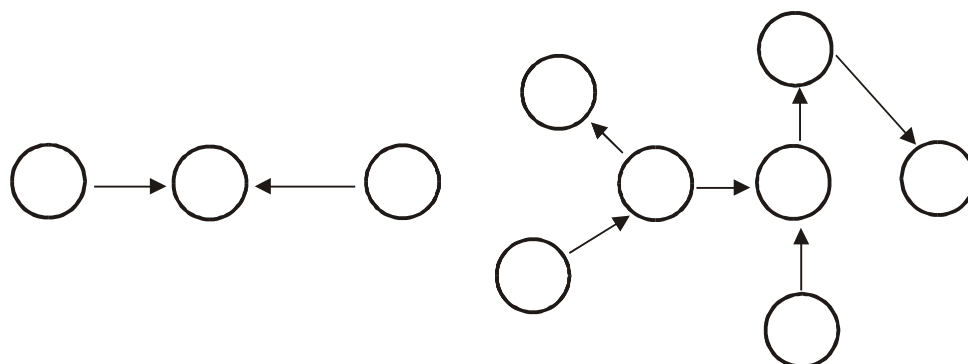


Figura 1.7. Árboles.

En un árbol enraizado se suelen definir relaciones de precedencia. Por ejemplo, en la Figura 1.8, el nodo 4 es el predecesor de los nodos 5 y 6, y el nodo 1 es el predecesor de los nodos 2 y 4. Cada nodo i, excepto el nodo raíz, tiene un único predecesor. Utilizaremos para almacenar el predecesor de un nodo i un índice de predecesor $Pred(i)$. Si $j= Pred(i)$, diremos que el nodo j es el predecesor del nodo i y este es un sucesor del nodo j.

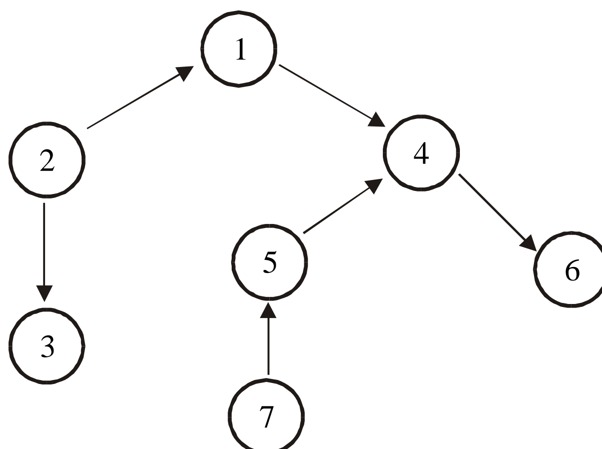


Figura 1.8. Árbol enraizado.

Árbol dirigido saliente: un árbol es un árbol dirigido saliente enraizado en un nodo s si, mediante un camino, se alcanza a todo nodo del árbol desde la raíz s.

Árbol dirigido entrante: un árbol es un árbol dirigido entrante enraizado en un nodo s si a través de un camino se alcanza a s desde cualquier nodo del árbol.

Árbol generador: un árbol T es un árbol generador del grafo G si, además, T es un subgrafo generador de G. Todo árbol generador de un grafo conexo G de n nodos tiene (n-1) arcos.

Ciclos fundamentales: Sea T un árbol generador del grafo G. La adición a T de cualquier arco de G que no está en T crea, exactamente, un ciclo. Denominaremos a tal ciclo, ciclo fundamental de G con respecto al árbol T. Dado un grafo de m arcos y n nodos y un árbol generador T, se tienen m-n+1 ciclos fundamentales.

Cortes fundamentales: Sea T un árbol generador del grafo G. Si quitamos cualquier arco de T producimos un grafo no conexo formado por dos subárboles. Aquellos arcos cuyos extremos pertenecen a diferentes subárboles constituyen un corte. Denominaremos a tal corte, corte fundamental. Dado un árbol generador T, hay n-1 cortes fundamentales.

Recorrido en amplitud de un grafo (***BFS***): Dado un grafo G, esta operación parte de un nodo i, y visita a todos sus vecinos contenidos en la lista de adyacencia $Suc(i)$ almacenándolos en una cola. Posteriormente toma como nuevo i el primer elemento de la cola y realiza la misma operación, hasta que todos los nodos del grafo resultan visitados o hasta que no se pueden visitar más. En el caso de utilizar la lista de adyacencia $Pred(i)$, hablamos de BFS inverso.

Recorrido en profundidad de un grafo (***DFS***): Dado un grafo G, esta operación parte de un nodo i, y visita a uno de sus vecinos j contenido en la lista de adyacencia $Suc(i)$. Posteriormente, se hace i=j y realiza la misma operación. Si un nodo no tiene vecinos o si los tiene pero han sido visitados, el nodo i vuelve a ser el vecino desde el que se alcanzó al propio i y se sigue recorriendo su lista de adyacencia en busca de un nodo no visitado. En el caso de utilizar la lista de adyacencia $Pred(i)$, hablamos de DFS inverso.

2.1 Representaciones de una red

En general, la eficacia de los procedimientos desarrollados para resolver problemas sobre redes depende de la representación computacional de estas. Una representación adecuada de la red mejora a menudo el tiempo de ejecución de un algoritmo. En este

apartado veremos algunas maneras de representar una red. En dicha representación se necesitan almacenar dos tipos de información: (1) la topología de la red, es decir, la estructura de los nodos y arcos; y (2) datos tales como costes, capacidades y ofertas/demandas asociados con los arcos y nodos de la red. Usualmente el esquema utilizado para representar la topología de la red sugiere formas naturales para almacenar la información asociada a los nodos y los arcos.

2.1.1 Matriz de Incidencia Nodo-Arco

La matriz de incidencia Nodo-Arco almacena la red mediante una matriz N de dimensión nxm, la cual contiene una fila por cada nodo y una columna por cada arco. La columna correspondiente al arco (i, j) tiene únicamente dos elementos distintos de cero: tiene un $+1$ en la fila correspondiente al nodo i y un -1 en la fila correspondiente al nodo j. De esta manera, la matriz de incidencia N se define de la manera siguiente:

$$N_{ik} = \begin{cases} +1 & \text{Si } i \in \text{V es el nodo cola del arco } a_k \in A \\ -1 & \text{Si } i \in V \text{ es el nodo cabeza del arco } a_k \in A \\ 0 & \text{en otro caso} \end{cases}$$

Se observa que de las nm entradas de la matriz, únicamente $2m$ de ellas son distintas de cero. Debido a esto, la representación de la matriz de incidencia es ineficiente. Sin embargo, está representación es importante ya que contiene la matriz de restricciones del problema fundamental de flujos en redes y porque se derivan varias propiedades teóricas interesantes.

2.1.2 Matriz de Adyacencia Nodo-Nodo

La matriz de adyacencia Nodo-Nodo almacena la red mediante una matriz M de dimensión nxn que tiene una fila y una columna para cada nodo. Cada elemento M_{ij} es igual a 1 si $(i, j) \in A$, e igual a cero en otro caso, es decir:

$$M_{ij} = \begin{cases} 1 & \text{si } (i, j) \in A \\ 0 & \text{en otro caso} \end{cases}$$

Si deseamos almacenar los costes y capacidades de los arcos, además de la topología de la red, podemos almacenar esta información en dos matrices adicionales de dimensión nxn.

La matriz de adyacencia tiene n^2 elementos, de los cuales únicamente m son distintos de cero. Esta representación es eficiente en espacio sólo si la red es lo suficientemente densa. Por otro lado, la simplicidad de esta representación permite implementar fácilmente determinados algoritmos de resolución de problemas sobre redes.

2.1.3 Listas de Adyacencia

Esta representación almacena la lista de adyacencia de cada nodo mediante una lista enlazada. Una lista enlazada es una colección de celdas, cada una conteniendo uno o más campos. La lista de adyacencia del nodo i será una lista enlazada conteniendo δ_i^+ celdas, donde cada celda corresponde a un arco $(i,j) \in A$. La celda correspondiente al arco $(i,j) \in A$ tendrá tantos campos como cantidad de información deseemos almacenar. Uno de los campos a almacenar es el nodo j. Otros dos campos podrían ser usados para almacenar el coste y la capacidad del arco. Cada celda contiene un campo adicional, llamado enlace, el cual almacena la posición de memoria (puntero) de la siguiente celda en la lista de adyacencia de este nodo. En el caso de que el siguiente elemento de la lista de adyacencia sea el vacío, el puntero contendrá el valor nulo.

Esta representación comprende n listas enlazadas, una por cada nodo. De esta manera, necesitamos un vector de punteros que enlacen con la primera celda de cada lista. La Figura 1.9 ilustra un ejemplo de esta representación.

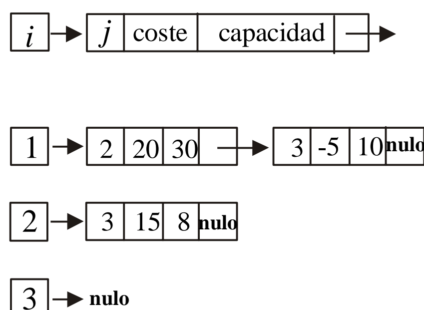


Figura 1.9. Lista de Adyacencia.

En muchos problemas de flujos en redes, cuando actualizamos alguna información acerca de un arco (i,j) también es necesario actualizar la información sobre el arco (j,i). Para realizar estas operaciones eficientemente debemos conocer, sobre la lista de adyacencia, dónde se encuentra el arco inverso (j,i) de cada arco (i,j). Para ello, se define un campo adicional "inverso", que contiene la posición de memoria que almacena la información del arco inverso. Así, el inverso del arco (i,j) apunta a la celda del arco (j,i) y al revés.

3. Problemas de flujos en redes y transformaciones

Consideremos una red dirigida $G=(V,A)$. Un flujo en G es un vector de variables $x=\{x_{ij}\}$ de m componentes, cada una asociada con un arco. Supongamos que cada arco $(i,j)\in A$ tiene asociado un coste c_{ij} que denota el coste por unidad de flujo sobre el arco. Asumiremos que el coste del flujo varía linealmente con la cantidad de flujo. Asociaremos con cada arco $(i,j)\in A$ una capacidad u_{ij} que denota la máxima cantidad de flujo que puede soportar dicho arco y una cota inferior l_{ij} que denota la mínima cantidad de flujo que debe pasar por el arco. Con cada nodo $i\in V$ asociaremos un valor entero b_i que representa la disponibilidad de dicho nodo. Si $b_i>0$, el nodo i es un nodo oferta, si $b_i<0$, el nodo i es un nodo demanda y si $b_i=0$, el nodo i es un nodo trasbordo. Las variables de decisión del problema son los flujos x_{ij} sobre cada arco $(i,j)\in A$. El *problema de flujo de coste mínimo* tiene la siguiente formulación:

$$\text{minimizar} \sum_{(i,j)\in A} c_{ij}x_{ij} \qquad (1.1a)$$

Sujeto a

$$\sum_{j\in Suc(i)} x_{ij} - \sum_{j\in Pred(i)} x_{ji} = b_i \,, \forall i\in V \qquad (1.1b)$$

$$l_{ij} \le x_{ij} \le u_{ij} \,, \forall (i,j)\in A \qquad (1.1c)$$

donde $\sum_{i=1}^{n} b_i = 0$. Llamaremos a (1.1b) *restricciones de conservación de flujo* y a (1.1c) *restricciones de acotación*. Para garantizar optimalidad finita, se asume que si $\{(i_1, i_2), ..., (i_k, i_1)\}$ es un ciclo en *G,* entonces debe satisfacer:

$$1)\ c_{i_1,i_2} + ... + c_{i_k,i_1} \geq 0 \ \text{ ó}$$

$$2)\ \min\ \{u_{i_1,i_2}, ..., u_{i_k,i_1}\} < \infty$$

Si no se cumple ni 1) ni 2), a la vez, entonces, el planteamiento del problema haría que, repetidamente, circule flujo a lo largo de este ciclo infinitas veces y, por tanto, se produzca la no acotación del mismo.

Del modelo de Flujo de Coste Mínimo derivan problemas importantes de Análisis de Redes como son el Problema de Flujo Máximo, el problema de Transporte, el Problema de Asignación, el Problema de Camino Mínimo,... También, la generalización o la ampliación de los términos que definen el anterior problema posibilitan la introducción de problemas con flujos múltiples, problemas de flujos multicriterio, problemas de flujos convexos, problemas de redes generalizadas, etc..

En la formulación de los problemas de flujos en redes, podemos adoptar uno de dos modelos equivalentes: podemos definir flujos sobre los arcos o flujos sobre caminos y ciclos. Por ejemplo, la formulación arco-flujo mostrada en la Figura 1.10a) envía 7 unidades de flujo desde el nodo 1 al nodo 6. En la Figura 1.10b) se muestra la equivalencia entre la formulación camino-flujo y la de arco-flujo. En esta formulación, se envían 4 unidades de flujo a lo largo del camino 1-2-4-6, 3 unidades de flujo a lo largo del camino 1-3-5-6 y 2 unidades de flujo a lo largo del ciclo 2-4-5-2.

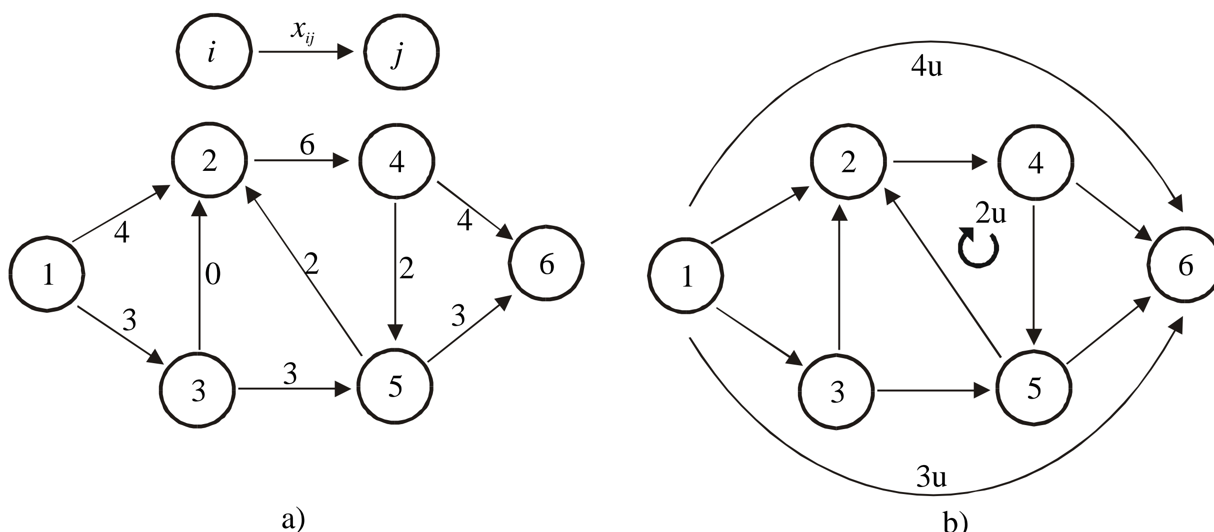


Figura 1.10. Formulación arco-flujo vs. Camino-flujo

Para establecer formalmente la equivalencia entre las dos maneras de expresar los flujos en una red, cambiaremos la restricción (1.1b) por la siguiente:

$$\sum_{j \in Suc(i)} x_{ij} - \sum_{j \in Pred(i)} x_{ji} = -\overline{b}(i), \forall i \in V \quad (1.1b')$$

donde $\sum_{i=1}^{n} \overline{b}(i) = 0$. Llamamos $\overline{b}(i)$ al balance del nodo i. El término $\overline{b}(i)$ representa la diferencia entre el flujo entrante y saliente del nodo i. Si $\overline{b}(i)>0$, decimos que el nodo i es un nodo de exceso; si $\overline{b}(i)<0$, decimos que el nodo i es un nodo de déficit y si $\overline{b}(i)=0$, i es un nodo balanceado.

En la formulación arco-flujo, las variables de decisión son los flujos x_{ij} sobre los arcos $(i,j) \in A$. La formulación de caminos y ciclos de flujo empieza numerando todos los caminos p entre cualquier par de nodos y todos los circuitos w de la red. Denotamos por P la colección de todos los caminos y por W la colección de todos los circuitos. Las variables de decisión, en la formulación camino-flujo, son $f(p)$ (el flujo sobre el camino p) y $f(w)$ (el flujo sobre el circuito w). Definimos esas variables para todo camino $p \in P$ y todo circuito $w \in W$.

Debemos notar que los conjuntos de caminos y circuitos de flujo determinan flujos sobre arcos: el flujo x_{ij} sobre el arco $(i,j) \in A$ es igual a la suma de los flujos $f(p)$ y $f(w)$ de todos los caminos p y

circuitos w que contienen a este arco. Para ilustrar esta idea es necesario introducir una notación adicional: $\delta_{ij}(p)$ es igual a 1 si el arco (i,j) esta contenido en el camino p e igual a 0 en otro caso. De manera similar, $\delta_{ij}(w)$ es igual a 1 si el arco (i,j) esta contenido en el ciclo w e igual a 0 en otro caso. Entonces,

$$x_{ij} = \sum_{p \in P} \delta_{ij}(p) f(p) + \sum_{w \in W} \delta_{ij}(w) f(w)$$

De esta forma, cada flujo sobre caminos y circuitos determina unívocamente flujos sobre arcos. Una pregunta oportuna es: ¿podemos descomponer flujos sobre arcos en flujos sobre caminos y circuitos?. El siguiente teorema nos suministra la respuesta.

Teorema 1.1. (*Teorema de descomposición del flujo*). *Todo flujo sobre caminos y circuitos puede ser representado unívocamente mediante flujos no negativos sobre arcos. Todo flujo no negativo sobre arcos puede ser representado mediante flujos sobre caminos y ciclos* (*no necesariamente de manera única*) *utilizando las siguientes propiedades.*

a) *Todo camino dirigido con flujo positivo conecta un nodo déficit con un nodo de exceso.*

b) *Como máximo n+m caminos y circuitos tienen flujo distinto de cero; y de ellos a lo sumo m son circuitos que tienen un flujo distinto de cero.*

(ver Ahuja et al.[4])

En el caso de una circulación, es decir, un flujo x para el que $\bar{b}(i)=0$ para todo $i \in V$, aplicando el teorema anterior tenemos que una circulación se descompone en flujos a lo largo de un máximo de m circuitos.

A continuación introducimos el concepto de ciclos incrementales con respecto a un flujo x. Un ciclo w (no necesariamente dirigido) en G es llamado ciclo incremental con respecto al flujo x si, enviando una cantidad de flujo positiva $f(w)$ alrededor del mismo, el flujo permanece factible. Esta consideración implica incrementar el flujo sobre los arcos hacia delante en el ciclo w y disminuir el flujo sobre los correspondientes

arcos hacia atrás. De esta manera, un ciclo w es incremental en G si $x_{ij} < u_{ij}$ para todos lo arcos (i,j) hacia delante y $x_{ij} > 0$ para todos los arcos (i,j) hacia atrás. Necesitamos extender la notación $\delta_{ij}(w)$ para los ciclos. Definimos $\delta_{ij}(w)$ igual a 1 si el arco (i,j) es un arco hacia delante en el ciclo w, igual a -1 si es un arco hacia atrás, e igual a 0 en otro caso. Definimos el coste de un ciclo incremental w como $c(w) = \sum_{(i,j)\in w} c_{ij}\delta_{ij}(w)$, donde c_{ij} es el coste unitario por unidad de flujo que atraviesa dicho arco. Así, el coste de un ciclo incremental representa el cambio en el coste de la solución factible si enviamos una unidad de flujo a lo largo del ciclo. El coste de enviar $f(w)$ unidades de flujo a lo largo del ciclo w es $c(w)f(w)$.

Teorema 1.2. (*Teorema del ciclo incremental*). *Sean x y x^0 cualesquiera dos soluciones factibles de un problema de flujos en redes. Entonces x es igual a x^0 más el flujo, como máximo, a través de m circuitos en $G(x^0)$. Además, el coste de x es igual al coste de x^0 más el coste de esos ciclos incrementales*. (ver Ahuja et al.[4])

3.1 Transformaciones de la red

Frecuentemente se hace necesario transformar las redes para simplificarlas, para mostrar equivalencia entre diferentes problemas de redes o para representar un problema de redes en la forma estándar requerida por un determinado código. En este apartado describiremos alguna de esas importantes transformaciones.

3.1.1 Cambio de arcos no dirigidos por arcos dirigidos

Algunas veces el problema de flujo de coste mínimo contiene arcos no dirigidos. Un arco no dirigido (i,j) con coste $c_{ij} \geq 0$ y capacidad u_{ij} permite enviar flujo desde el nodo i al nodo j y también desde el nodo j al nodo i; una unidad en cualquier dirección cuesta c_{ij} y el flujo total esta acotado por u_{ij}. Es decir, el modelo no dirigido tiene como restricción $x_{ij} + x_{ji} \leq u_{ij}$ y los términos $c_{ij}x_{ij} + c_{ij}x_{ji}$ en la función objetivo. Ya que el coste $c_{ij} \geq 0$, en la

solución óptima una de las variables x_{ij} y x_{ji} será cero. En este caso diremos que la solución es no solapada.

En la subsiguiente discusión nos referiremos al arco no dirigido (i,j) como $\{i,j\}$. Asumiremos que el flujo del arco en cualquier dirección de $\{i,j\}$ tiene una cota inferior igual a cero; la transformación que consideraremos no es válida si el flujo del arco tiene una cota inferior distinta de cero o el coste del arco es negativo. Para transformar el arco no dirigido al caso dirigido, reemplazamos cada arco no dirigido $\{i,j\}$ por dos arcos dirigidos, (i,j) y (j,i), ambos con coste c_{ij} y capacidad u_{ij}. Para establecer el funcionamiento de esta transformación, mostraremos que todo flujo no solapado en la red original tiene un flujo asociado en la red transformada con el mismo coste y al revés. Si el arco no dirigido $\{i,j\}$ lleva α unidades de flujo desde el nodo i al j, en la red transformada $x_{ij}=\alpha$ y $x_{ji}=0$. Si el arco no dirigido $\{i,j\}$ lleva α unidades de flujo desde el nodo j al i, en la red transformada $x_{ij}=0$ y $x_{ji}=\alpha$. Por otro lado, si x_{ij} y x_{ji} son los flujos sobre los arcos (i,j) y (j,i) en la red dirigida, $x_{ij}-x_{ji}$ o $x_{ji}-x_{ij}$ es el flujo positivo asociado con el arco $\{i,j\}$ en la red no dirigida. Si $x_{ij}-x_{ji}$ es positivo, el flujo desde el nodo i al nodo j sobre el arco $\{i,j\}$ es igual a esta cantidad. Si $x_{ji}-x_{ij}$ es positivo, el flujo desde el nodo j al nodo i sobre el arco $\{i,j\}$ es igual a esta cantidad. En cualquier caso, el flujo en la dirección opuesta es cero (no solapado). Si $x_{ij}-x_{ji}=x_{ji}-x_{ij}$ es cero, el flujo sobre el arco $\{i,j\}$ es cero.

3.1.2 Eliminación de cotas inferiores distintas de cero

Si un arco (i,j) tiene una cota inferior l_{ij} distinta de cero sobre el flujo x_{ij}, podemos reemplazar x_{ij} por $x'_{ij}+l_{ij}$ en la formulación del problema. Ahora las restricciones de acotación se transforman en $l_{ij}\le x'_{ij}+l_{ij}\le u_{ij}$ o $0\le x'_{ij}\le u_{ij}-l_{ij}$. Realizando esta sustitución en las restricciones de conservación de flujo, disminuimos b_i en l_{ij} unidades e incrementamos b_j en l_{ij} unidades. Esta sustitución cambia el valor de la función objetivo en un valor constante que

podemos almacenar separadamente e ignorarlo en la resolución del problema.

3.1.3 Inversión de arcos

La transformación de inversión de arcos es utilizada para eliminar arcos con coste negativo. En esta transformación reemplazamos la variable x_{ij} por $u_{ij} - x_{ij}$. Haciendo esto, sustituimos el arco (i,j), que tiene un coste asociado c_{ij}, por el arco (j,i) con un coste asociado $-c_{ij}$. La Figura 1.11 muestra esta transformación. Primero enviamos u_{ij} unidades de flujo sobre el arco, lo que disminuye b_i en u_{ij} unidades y aumenta b_j en u_{ij} unidades. La nueva variable x_{ji} mide la cantidad de flujo que podemos eliminar de la capacidad completa de flujo u_{ij}.

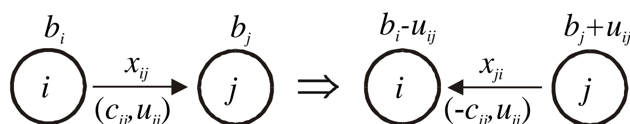


Figura 1.11. Inversión de arcos.

3.1.4 Eliminación de las capacidades de los arcos

Si un arco (i,j) tiene una capacidad positiva u_{ij}, podemos eliminar la capacidad usando la transformación siguiente. Introducimos un nodo adicional de tal manera que las restricciones de capacidad del arco (i,j) aparezcan como restricciones de conservación de flujo del nuevo nodo. Supongamos que introducimos la variable de holgura $s_{ij} \geq 0$ y escribimos la desigualdad $x_{ij} \leq u_{ij}$ como la igualdad $x_{ij} + s_{ij} = u_{ij}$. Multiplicando ambos lados por -1, obtenemos $-x_{ij} - s_{ij} = -u_{ij}$. Tratamos a esta igualdad como una restricción más de conservación de flujo. Esta manipulación algebraica corresponde a la transformación de la red mostrada en la Figura 1.12.

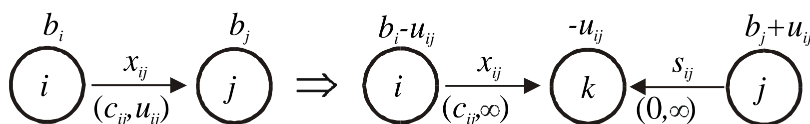


Figura 1.12. Eliminación de capacidades.

Para ver la relación entre los flujos de la red original y transformada, realizamos las siguientes observaciones. Si x_{ij} es el flujo sobre el arco (i,j) en la red original, en la red transformada se tiene que $x'_{ik} = x_{ij}$ y $x'_{jk} = u_{ij} - x_{ij}$. Podemos notar que ambos flujos x y x' tienen el mismo coste. De la misma manera, x'_{ik} y x'_{jk}, en la red transformada producen un flujo $x_{ij} = x'_{ik}$ de mismo coste que en la red original. Además, ya que $x'_{ik} + x'_{jk} = u_{ij}$ y que x'_{ik} y x'_{jk} son no negativos, tenemos que $x_{ij} = x'_{ik} \le u_{ij}$. Así, el flujo x_{ij} satisface la capacidad del arco y la transformación modela correctamente las capacidades de los arcos.

3.1.5 Redes residuales

En el diseño, desarrollo e implementación de algoritmos de flujos en redes, a menudo es conveniente medir el flujo no en términos absolutos sino en términos incrementales con respecto a una solución factible dada. Para ello introducimos el concepto de *red residual* la cual funciona como una red de flujo remanente que puede llevar flujo adicional. El concepto de red residual esta basado en la siguiente idea intuitiva. Si se supone que el arco (i,j) lleva x_{ij} unidades de flujo, entonces podemos enviar $u_{ij} - x_{ij}$ unidades de flujo adicional desde el nodo i al nodo j. Debemos notar también que podemos enviar x_{ij} unidades de flujo desde el nodo j al nodo i a través del arco (i,j), lo que cancelaría el flujo existente sobre el arco. Además, enviar una unidad de flujo desde el nodo i al nodo j sobre el arco (i,j) incrementa el coste del flujo en c_{ij} unidades. Por el contrario, enviando flujo desde el nodo j al nodo i sobre el mismo arco disminuye el coste en c_{ij} unidades.

Teniendo en cuenta esto, definimos la red residual con respecto al flujo x como sigue: Consideramos una red en la que reemplazamos cada arco (i,j) en la red original por dos arcos (i,j) y (j,i). El arco (i,j) tiene un coste igual a c_{ij} y una *capacidad residual* $r_{ij} = u_{ij} - x_{ij}$ y el arco (j,i) tiene un coste igual a $-c_{ij}$ y una capacidad residual $r_{ji} = x_{ij}$. La Figura 1.13 ilustra esta definición. La red residual contiene únicamente arcos con capacidades residuales

positivas. Usaremos la notación $R = G(x)$ para denotar la red residual correspondiente al flujo x.

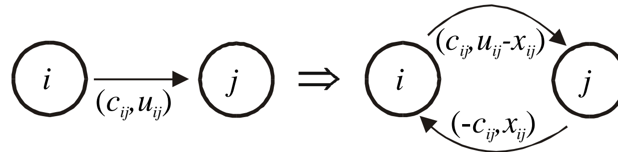


Figura 1.13. Red Residual

El concepto de red residual puede implicar la revisión de decisiones previas, ya que, *pasar de un patrón de flujo a otro, mejorando la función objetivo, requiere, a veces, la cancelación de envíos de flujos anteriores*.

4. Complejidad computacional

Antes de plantear los distintos problemas de flujos en redes, introducimos los conceptos necesarios para analizar la eficiencia de un algoritmo. Todos los algoritmos que resuelven un determinado problema pueden no resultar iguales porque, por ejemplo, alguno de ellos sea más rápido que otros. Por lo tanto, nos hallamos en la necesidad de encontrar patrones que indiquen cuándo un algoritmo es mejor que otro y por qué. Para ello introduciremos el concepto de complejidad computacional.

De una forma poco precisa podemos decir que un algoritmo es un conjunto de operaciones detalladas y no ambiguas, a ejecutar paso a paso, que conducen a la resolución de un problema. Generalmente, estamos interesados en encontrar el algoritmo más eficiente para resolver un problema. Usaremos el tiempo empleado por el algoritmo como medida de la eficiencia del mismo, aunque en general deben tenerse en cuenta los recursos que emplea para obtener la solución. Definimos *instance* al caso particular de un problema con datos específicos para todos los parámetros del problema. Hemos de tener en cuenta que un algoritmo podrá resolver rápidamente determinados casos particulares de un problema y tardar demasiado en la resolución de otros.

Las diferentes instrucciones típicas de un algoritmo son instrucciones de asignación, aritméticas y lógicas. El número total de las mismas resulta de la suma de todas las instrucciones definidas anteriormente y determina el tiempo requerido en la ejecución del algoritmo.

La literatura especializada tiene, aceptados ampliamente, tres enfoques básicos para la medida de la bondad de un algoritmo:

Análisis empírico. El objetivo del análisis empírico es estimar cómo funcionan los algoritmos en la práctica. En este análisis es habitual desarrollar un programa de ordenador y medir la bondad del programa para distintas clases de instances.

Análisis caso-promedio. El objetivo del análisis caso-promedio consiste en estimar el número de pasos esperados que realiza un algoritmo. En este análisis elegimos una distribución de probabilidad sobre los instances y, usando algún análisis estadístico, obtenemos el tiempo asintótico de ejecución para el algoritmo.

Análisis del caso peor. El análisis del caso peor suministra una cota superior del número de pasos que un algoritmo realiza para algún instance. En este análisis se cuenta el mayor número de pasos posibles necesitados para la resolución del problema.

Cada una de estas tres medidas tienen sus ventajas e inconvenientes. El análisis empírico tiene algunas desventajas: 1) el funcionamiento de un algoritmo depende del lenguaje de programación, compilador y computador usado para las experiencias computacionales, así como del programador que escribió el programa; 2) a menudo este tipo de análisis consume mucho tiempo; y 3) la comparación de los algoritmos es a menudo poco concluyente en el sentido de que diferentes algoritmos funcionan mejor frente a diferentes clases de problemas y diferentes estudios computacionales establecen resultados contradictorios.

El análisis caso-promedio tiene también desventajas: 1) el análisis depende crucialmente de la distribución de probabilidad

elegida para representar los problemas particulares y diferentes elecciones podrían conducir a diferentes consideraciones de los méritos relativos de los diferentes algoritmos bajo consideración; 2) a menudo es difícil determinar apropiadamente la distribución de probabilidad de los problemas considerados en la práctica; y 3) el análisis requiere, a menudo, cálculos matemáticos complicados para evaluar los tipos más simples de algoritmos, pudiéndose complicar en grado sumo cuando los algoritmos son complejos. Además, para realizar este tipo de análisis, se necesitan resolver un gran número de problemas particulares, pudiendo suceder que el funcionamiento de un algoritmo sea bueno salvo para contadas excepciones, en las que se comporta muy mal, contribuyendo significativamente al estudio estadístico.

El análisis del caso peor evita muchos de los anteriores inconvenientes. Este análisis es independiente del entorno computacional, es relativamente fácil de realizar y es definitivo en el sentido de que suministra conclusiones que permiten asegurar que un algoritmo es mejor que otro para el problema peor posible que pudiéramos encontrar. Pero el análisis del caso peor no está exento de inconvenientes. Uno de sus mayores desventajas es que permite utilizar instances patológicos para determinar la eficiencia del algoritmo, aún cuando estos pudieran ser sumamente raros en la práctica. Por otro lado, este tipo de análisis garantiza una cota superior del número de pasos que puede realizar el algoritmo pero esconde información acerca de la resolución de casos no tan extremos, es decir, un algoritmo podría requerir para la resolución de muchos de sus problemas un número de pasos muy inferior al determinado por el análisis del caso peor.

Para estimar la bondad de los distintos algoritmos, en este trabajo utilizaremos el análisis del caso peor, aunque estamos convencidos de que un estudio experimental de los algoritmos suministra información importante para guiarnos en su utilización práctica, dependiendo del instance del problema que se ha de resolver.

4.2 Tamaño de un problema

Frecuentemente, el esfuerzo requerido para resolver un problema varía desigualmente con su tamaño. Dado el tamaño de un dato cuyo valor es q, podemos realizar dos posibles hipótesis: a) asumir que el tamaño del dato es q, b) asumir que el tamaño del dato es $\log(q)$. La primera consideración se conoce como criterio de coste uniforme y considera que el espacio requerido para almacenar q es proporcional a su valor. La segunda, conocida como criterio de coste logarítmico, considera que ya que la representación binaria del valor q requiere $\lceil \log(q) \rceil$ bits, el espacio requerido para almacenar q es proporcional a $\log(q)$.

El tamaño de un problema de redes es una función de como ha sido establecido. Supongamos que especificamos la red mediante la representación de listas de adyacencia, la cual es la representación más eficiente en espacio que podemos usar. Entonces, el tamaño del problema es el número de bits empleados para almacenar estas listas de adyacencia. Como en esta representación se almacena un puntero para cada nodo y cada arco, y un elemento para cada valor del coste y capacidad del arco, se requieren, aproximadamente, $n \cdot \log(n) + m \cdot \log(m) + m \cdot \log(C) + m \cdot \log(U)$ bits para almacenar todos los datos del problema de flujo de coste mínimo, donde $U = \max \{u_{ij} / (i,j) \in A\}$ y $C = \max \{|c_{ij}| / (i,j) \in A\}$.

En principio, podríamos representar el tiempo de ejecución de un algoritmo como función del tamaño del problema, es decir, del número de bits necesarios para representar el instance. Sin embargo, esto podría ser, innecesariamente, inoportuno. Expresaremos el tiempo de ejecución de un algoritmo, de forma más simple y directa, como una función de los parámetros de la red n, m, $\log(C)$ y $\log(U)$.

4.3 Complejidad del caso peor

El tiempo de ejecución de un algoritmo depende de la naturaleza y tamaño de la entrada. Una función temporal de la complejidad de un algoritmo es una función del tamaño del problema y especifica el tiempo máximo que se necesita para

resolver un instance de un tamaño dado. En otras palabras, la función temporal de complejidad da una medida de la proporción en que se incrementa el tiempo de resolución con respecto a un incremento en el tamaño del problema. Nos referiremos a la función temporal de complejidad del caso peor de un algoritmo como su cota del caso peor (una cota superior del tiempo invertido).

4.4 Notación O grande

Para definir la complejidad total de un algoritmo necesitamos especificar los valores para una o más constantes. En muchos casos, la determinación de esas constantes es una tarea no trivial. Precisamente, su determinación podría depender del computador y de otros factores. Consideremos, como ejemplo, el siguiente segmento de un programa que suma dos matrices de orden pxq:

```
for i := 1 to p do
    for j:=1 to q do
        c_ij = a_ij + b_ij;
```

En un primer vistazo, vemos que el programa realiza pq sumas y el mismo número de asignaciones de valores que el computador almacenará en las variables c_{ij}. Este primer vistazo, ignora, muchas operaciones que el ordenador debe realizar. Un ordenador generalmente almacena 2 matrices de tamaño pxq como un simple vector de longitud pq y, así, el elemento a_{ij} será almacenado en la celda $(i\text{-}1)q+j$ del vector a. Por ello, cada vez que accedemos a los valores a_{ij} y b_{ij} necesitamos, para cada uno de ellos, realizar una resta, una multiplicación y una suma. Además, siempre que el ordenador incrementa el índice i ó j, deberá realizar una comparación para determinar si $i>p$ ó $j>q$, respectivamente. Por lo tanto, un análisis detallado de un algoritmo, a tan bajo nivel, consume mucho tiempo y no es particularmente aclaratorio.

La dependencia de la función de complejidad de las constantes tiene todavía otros problemas: ¿Cómo comparamos un algoritmo que realiza $5n$ sumas y $3n$ comparaciones con un algoritmo que realiza n multiplicaciones y $2n$ restas?. Diferentes computadoras realizan operaciones aritméticas y lógicas a distintas velocidades. Debido a esto, ninguno de esos algoritmos será universalmente el mejor.

Evitaremos estas dificultades ignorando las constantes en el análisis de la complejidad. Usaremos la notación *O grande* para remplazar expresiones como: “el algoritmo requiere un tiempo cnm para alguna constante c", por la expresión equivalente: “el algoritmo requiere un tiempo O(nm)”. Formalizaremos la definición como sigue:

Un algoritmo se dice que tiene un tiempo de computación de O($f(n)$) si para algunos c y n_O, el tiempo invertido por el algoritmo es como mucho $cf(n)$ para todo n mayor o igual que n_O.

Una vez establecida la definición en términos de n, podemos, fácilmente, incorporar otros parámetros como son m, C y U en la definición.

La notación O grande tiene unas cuantas implicaciones. La complejidad de un algoritmo es una cota superior del tiempo de ejecución del algoritmo para valores de n lo suficientemente grandes. Más aún, esta notación indica sólo los términos más dominantes en el tiempo de ejecución, y representa el hecho de que para un n grande, los términos con un crecimiento menor tienden a ser insignificantes si los comparamos con el término de mayor crecimiento. Por ejemplo, si el tiempo de ejecución de un algoritmo es $100n+n^2+0{,}0001\,n^3$, entonces para todo $n > 100$ el segundo término domina al primer término y para todo $n > 10000$ el tercer término domina al segundo término. Así, la complejidad del algoritmo es O(n^3).

Otra importante implicación de ignorar las constantes en el análisis de la complejidad es que asumiremos que las operaciones elementales, tales como la suma, resta, multiplicación, división, asignación y operaciones lógicas, requieren una misma cantidad de tiempo.

4.5 Algoritmos polinomiales y exponenciales

Una idea que ha ganado en aceptación en los últimos años es la de considerar que un algoritmo de redes es bueno si la complejidad del caso peor esta acotada por una función polinomial de los

parámetros del problema. Un algoritmo que cumpla esto es llamado *algoritmo polinomial*. Algunos ejemplos de cotas polinomiales son O(n^2), O(nm), O(m+$n$log(C)). Un algoritmo polinomial se dice que es *fuertemente polinomial* si su complejidad esta acotada por una función polinomial en n y m y no aparece log(C) ó log(U). En caso contrario se dice que el algoritmo es *débilmente polinomial*.

Se dice que un *algoritmo es exponencial* en tiempo, si su ejecución no puede ser acotada polinomialmente por la longitud de la entrada. Algunos ejemplos son O(2^n), O(n!) y O($n^{\log(n)}$). Diremos que un *algoritmo es pseudopolinomial* si su tiempo de ejecución es polinomialmente acotado en n, m, C y U. Algunos ejemplos son O(m+nC) y O(mC).

Hay razones para preferir algoritmos polinomiales a algoritmos exponenciales. Las funciones de complejidad exponencial tienen un crecimiento explosivo y en general resuelven solo pequeños problemas. En la Tabla 1.1 se ilustra el crecimiento de complejidades típicas:

n	log(n)	$n^{0.5}$	n^2	n^3	2^n	n!
10	3,62	3,16	10^2	10^3	10^3	$3{,}6 \cdot 10^6$
100	6,64	10,00	10^4	10^6	$1{,}27 \cdot 10^{30}$	$9{,}33 \cdot 10^{17}$
1000	9,97	31,62	10^6	10^9	$1{,}07 \cdot 10^{31}$	$4{,}02 \cdot 10^{27}$
1000	13,29	100,0	10^8	10^{12}	$0{,}99 \cdot 10^{30}$	$2{,}85 \cdot 10^{39}$

Tabla 1.1. Crecimiento de algunas complejidades típicas.

Debemos notar que las complejidades O(n^3) y O(2^n) para n=10 son, aproximadamente, la misma. También se puede ver que, para n<10, la complejidad O(2^n), a pesar de ser exponencial, es mejor que O(n^3).

4.6 Notación Ω y Θ grande

Como acabamos de ver, la notación O grande especifica una cota superior sobre el tiempo de ejecución de un algoritmo. La notación Ω grande especifica una cota inferior del tiempo de ejecución. Más precisamente:

Un algoritmo se dice que tiene un tiempo de computación de $\Omega(f(n))$ si para algunos c' y n_0, el tiempo invertido por el algoritmo es al menos $c'f(n)$ para todo n mayor o igual que n_0.

La notación Θ grande da una cota inferior y superior sobre el funcionamiento del algoritmo, es decir:

Un algoritmo se dice que es $\Theta(f(n))$ si el algoritmo es a la vez $O(f(n))$ y $\Omega(f(n))$.

4.7 Recuento de operaciones representativas

Como hemos comentado, el análisis del caso peor puede ser muy pesimista si utiliza instances patológicos para determinar la eficiencia de un algoritmo, aún cuando estos casos particulares se den con muy poca frecuencia. A menudo, el comportamiento empírico de un algoritmo es mucho más sugerente que su análisis del caso peor. Por esto, la comunidad investigadora valora frecuentemente la eficiencia práctica de un procedimiento teniendo en cuenta su comportamiento empírico.

La literatura existente sobre los tests computacionales tiene una tendencia excesiva a considerar el tiempo de CPU como primera medida de la eficiencia. El tiempo de CPU depende de un subconjunto de detalles del entorno computacional tales como (1) el lenguaje de programación elegido, compilador y computador; (2) el estilo del programador; (3) el generador de redes; (4) las combinaciones de los tamaños de la entrada,...,etc. Debido a todo esto, los tiempos de CPU son a menudo difíciles de replicar, lo que contradice el espíritu de la investigación científica. Hay que buscar medidas alternativas al tiempo de CPU. Por ejemplo, un algoritmo realiza generalmente algunas operaciones fundamentales repetidamente, y un típico análisis del tiempo de CPU no ayuda a identificar a esas operaciones bloqueantes. Identificando las operaciones bloqueantes de un algoritmo podemos encontrar información para dirigir los esfuerzos futuros para entenderlo y mejorarlo.

Podemos medir el comportamiento empírico de un algoritmo contando el número de veces que ejecuta cada una de esas operaciones bloqueantes mientras resuelve casos particulares del problema. Es decir, debemos dirigir la experiencia computacional en la obtención del número de operaciones bloqueantes realizadas en lugar de obtener una cota superior teórica de este número.

Supongamos que la ejecución de cualquier línea de código requiere un tiempo O(1) y que el código investigado tiene K líneas. Entonces, dado un instance I del problema, sea $\alpha_k(I)$, con k=1,...,K el número de veces que se ejecuta la línea k para el instance I. Denotemos por CPU(I) al tiempo de CPU en la resolución del instance I. Entonces, tenemos que

$$\text{CPU}(I)=\text{O}(\sum_{k=1}^{K}\alpha_k(I)).$$

Sea S un subconjunto de {1,...,K} y sea a_S el conjunto $\{a_i : i \in S\}$. Decimos que a_S es un conjunto de líneas de códigos representativas del programa, si para alguna constante c se tiene que $\alpha_i(I) \le c\left(\sum_{k\in S}\alpha_k(i)\right)$, para todo instance I del problema y para toda línea a_i del código. En otras palabras, el tiempo empleado en ejecutar la línea a_i esta dominado por el tiempo empleado en la ejecución de las líneas de código en a_S. Dada esta definición, tenemos que $\text{CPU}(I)=\text{O}(\sum_{k\in S}\alpha_k(I))$.

Por lo tanto, el objetivo en el análisis empírico de un algoritmo es identificar las operaciones representativas y, para ello, realizar un recuento del número de veces que son ejecutadas. Algunas veces estas operaciones no se refieren específicamente a una línea de código sino a una operación del algoritmo que utiliza un número constante de líneas de código.

Utilizaremos también el concepto de *tiempo de CPU virtual* para un instance I que denotamos por CPUV(I). El tiempo de CPU virtual de un algoritmo es una estimación lineal de su tiempo de CPU usando el recuento de las operaciones representativas, es

decir, CPUV(I)=$\sum_{k\in S} c_k \alpha_k(I)$ para un conjunto de constantes c_k con k=1,..., $|S|$, tales que CPUV(I) es la mejor estimación posible del tiempo de CPU(I) para el instance I. Una posible vía de determinación de las constantes puede consistir en usar análisis de regresión (múltiple). Para ello, consideramos los puntos (CPU(I), $\alpha_1(I),\ldots,\alpha_{|S|}(I)$) generados en la resolución de varios instances. Usaremos análisis de regresión para determinar las constantes que minimizan la expresión $\sum_{I}(CPU(I)-CPUV(I))^2$.

5. Problema de flujo de coste mínimo

Recordemos la notación asociada al problema de flujo de coste mínimo introducido con anterioridad. Dada una red dirigida G=(V,A) supongamos que cada arco $(i,j)\in A$ tiene asociado un coste c_{ij} que denota el coste por unidad de flujo sobre el arco. Asumiremos que el coste varía linealmente con la cantidad de flujo. Asociaremos con cada arco $(i,j)\in A$ una capacidad u_{ij} que denota la máxima cantidad de flujo que puede soportar dicho arco y una cota una cota inferior l_{ij} que denota la mínima cantidad de flujo que debe pasar por el arco. Con cada nodo $i\in V$ asociaremos un valor entero b_i que representa la oferta/demanda de dicho nodo. Las variables de decisión del problema son los flujos x_{ij} sobre cada arco $(i,j)\in A$. Definimos también:

$$\mathrm{C}= \max \{c_{ij} : (i,j)\in A\}$$
$$\mathrm{U}= \max \{\max \{|b_i| : i\in V\}, \max \{u_{ij} : (i,j)\in A \text{ y } u_{ij}<\infty\}\}$$

Asumiremos, mientras no se diga lo contrario, que todos los datos son enteros. supondremos, también, que el problema de flujo de coste mínimo satisface las dos siguientes condiciones:

a) ***Factibilidad*.** Se supone que $\sum_{i\in N} b_i = 0$ y que el problema de flujo de coste mínimo tiene una solución factible.

b) *Conexidad.* Asumiremos que la red G contiene un camino dirigido sin capacidades (es decir, con capacidad ∞) entre cada par de nodos de la red. Impondremos esta condición, si es necesario, añadiendo un nodo artificial, $n+1$, y arcos artificiales $(n+1,j)$ y $(j,n+1)$ para cada $j \in V$ y asignando un coste grande y una capacidad infinita a cada uno de esos arcos. Estos arcos no deberían aparecer en la solución óptima salvo que el problema no tenga solución factible.

Presentaremos a continuación algunos teoremas acerca del problema de flujo de coste mínimo. No entraremos en la demostración de los mismos.

Teorema 1.3. *Si todas las capacidades máximas y mínimas son enteras, entonces existe un flujo de coste mínimo cuyas componentes son todas enteras.*

Las condiciones de optimalidad (CO) para el problema de flujo de coste mínimo se dan en el siguiente teorema.

Teorema 1.4. *Sea x un flujo factible. Una condición necesaria y suficiente para que x sea óptimo es que exista un vector potencial $\pi \in \mathrm{R}^n$ tal que*:

$$c_{ij} - \pi_i + \pi_j > 0 \Rightarrow x_{ij} = l_{ij}$$
$$c_{ij} - \pi_i + \pi_j < 0 \Rightarrow x_{ij} = u_{ij}$$

Definimos el *coste reducido* de un arco (i,j) como $\bar{c}_{ij} = c_{ij} - \pi_i + \pi_j$. Por lo tanto el par (x, π) de flujos y potenciales es óptimo si satisface las siguientes condiciones: (1) x es factible, (2) si $\bar{c}_{ij} > 0 \Rightarrow x_{ij} = l_{ij}$, (3) si $\bar{c}_{ij} = 0 \Rightarrow l_{ij} \le x_{ij} \le u_{ij}$,(4) si $\bar{c}_{ij} < 0 \Rightarrow x_{ij} = u_{ij}$.

Estos resultados, relativos a la optimalidad de un problema de flujo de coste mínimo, son vitales a la hora de desarrollar algoritmos para su resolución.

A continuación, estudiaremos el Método Simplex para redes.

El algoritmo Simplex que utilizamos es una especialización del algoritmo Simplex Primal para Programación Lineal con variables acotadas.

Antes de desarrollar el algoritmo Simplex para redes, describiremos el concepto de *estructura básica*, introduciendo una estructura de datos para almacenar y manipular la base, la cual es un árbol generador. Dada una estructura básica, mostraremos cómo computar los flujos en los arcos y los potenciales de los nodos. Discutiremos la forma de realizar varias operaciones Simplex como la selección del arco entrante, del arco saliente y el pivoteo, usando la estructura de árbol. Finalmente, veremos cómo garantizar la finitud del algoritmo Simplex para redes.

El algoritmo Simplex para redes mantiene una solución básica factible primal en cada iteración. Una solución básica para el problema de flujo de coste mínimo se define por una tripleta (B,L,U); donde B, L y U es una partición del conjunto de arcos A de la red. El conjunto B denota el conjunto de *arcos básicos o de la base*, y L, U denotan, respectivamente, el conjunto de *arcos no básicos* en sus cotas inferiores y superiores. Nos referiremos a la tripleta (B,L,U) como la *estructura básica*.

Una estructura (B,L,U) se dice que es factible si $x_{ij} = l_{ij}$ para cada $(i,j) \in L$, $x_{ij} = u_{ij}$ para cada $(i,j) \in U$ y el problema tiene una solución factible satisfaciendo las restricciones de las ecuaciones de conservación del flujo y las restricciones de acotación sobre las variables x_{ij}. Una estructura básica factible (B,L,U) se dice que es una *estructura básica óptima* si es posible obtener un conjunto de potenciales de los nodos π tal que el coste reducido $\bar{c}_{ij} = c_{ij} - \pi_i + \pi_j$ satisfaga las siguientes condiciones de optimalidad:

$$\bar{c}_{ij} = 0, \ \forall (i,j) \in \mathrm{B}$$
$$\bar{c}_{ij} \geq 0, \ \forall (i,j) \in \mathrm{L}$$
$$\bar{c}_{ij} \leq 0, \ \forall (i,j) \in \mathrm{U}$$

El algoritmo Simplex para redes mantiene una estructura básica en cada iteración y, sucesivamente, modifica la estructura básica hasta que llegue a ser óptima.

La especialización del algoritmo Simplex para redes se fundamenta en que la base (los arcos en B) forman un árbol generador. El algoritmo requiere que el árbol sea representado de manera que las operaciones se realicen eficientemente, permitiendo actualizar adecuadamente la representación del árbol.

Consideraremos que el árbol *cuelga* de un nodo especial llamado *raíz*. Asumiremos que 1 es el nodo raíz. La Figura 1.14 muestra un ejemplo de árbol básico.

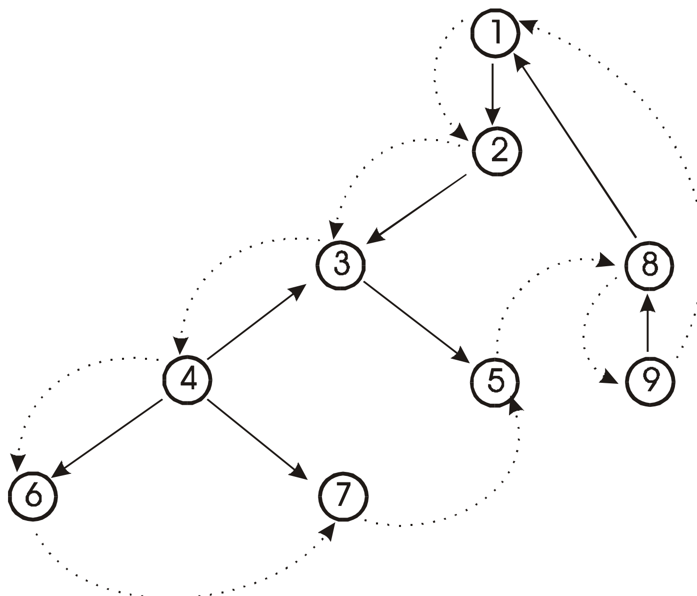


Figura 1.14. árbol generador básico.

Asociaremos tres índices con cada nodo i en el árbol: un índice del predecesor, $Pred(i)$; un índice de profundidad, $Depth(i)$; y un índice que indica el nodo que cuelga del nodo actual, $Thread(i)$. Para cada nodo i existe una única cadena en el árbol, que lo conecta con la raíz. El índice $Pred(i)$ almacena el primer nodo en ese camino y el índice $Depth(i)$ almacena el número de arcos en el camino. Para la raíz esos índices son cero. La Tabla 1.2 muestra los valores de estos índices para el árbol de la Figura 1.14. Hay que notar que, usando iterativamente el índice del predecesor, podemos enumerar el camino desde algún nodo a la raíz. Diremos que $Pred(i)$ es el *predecesor* del nodo i y que i es el *sucesor* del nodo $Pred(i)$. Los *descendientes* de un nodo i, consisten del nodo i, sus sucesores, los sucesores de sus sucesores, etc.. Un nodo que no tiene sucesores

se denomina nodo *hoja*. Por ejemplo, en la figura anterior los descendientes del nodo 3 son {3,4,5,6,7}, los nodos hojas son 6,7,5,9.

i	1	2	3	4	5	6	7	8	9
$Pred(i)$	0	1	2	3	3	4	4	1	8
$Depth(i)$	0	1	2	3	3	4	4	2	3
$Thread(i)$	2	3	4	6	8	7	5	9	1

Tabla 1.2. Índices del árbol generador básico.

Los índices *Thread* definen una secuencia de enumeración de los nodos del árbol que comienza en la raíz y visita los nodos en el orden de arriba a abajo y de izquierda a derecha y, finalmente, vuelve a la raíz. Esta secuencia forma, por tanto, un único ciclo que visita todos nodos exactamente una vez, con la excepción del nodo raíz. Los índices pueden ser computados realizando un recorrido en profundidad (DFS) en el árbol, donde $Thread(i)$ coincide con el nodo j que fue el último nodo marcado como visitado en el orden DFS.

5.2 Pseudo-código del Método Simplex para redes

El siguiente pseudo-código especifica los pasos esenciales del algoritmo:

```
Algoritmo Simplex_para_Redes;
  begin
  Determina un flujo x inicial, factible, y la correspondiente
  estructura básica (B,L,U);
  Computa los potenciales π  para esta estructura básica;
  while algún arco viole las condiciones de optimalidad do
    begin
    Selecciona un arco entrante (i,j) violando las condiciones
    de optimalidad;
    Añade el arco (i,j) al árbol generador a B;
    Identifica el ciclo formado y envía el máximo flujo posible
    a lo largo del mismo;
    Determina el arco saliente (p,q);
    Realiza el cambio de la base y actualiza los potenciales π
    end
  end;
```

A continuación, describiremos los pasos de este algoritmo con más detalle.

5.3 Obtención de una estructura básica inicial

La hipótesis de conexión permite la obtención de una solución básica factible. No obstante, asumiremos que para todo nodo $j \in V$ la red contiene los arcos ficticios $(n+1,j)$ ó $(j,n+1)$ con costes y capacidades suficientemente grandes, donde hemos añadido un nodo ficticio $n+1$ al que consideramos como el nodo raíz. La base inicial B incluye el arco $(n+1,j)$ con un flujo de $-b_j$, si $b_j \leq 0$ ó el arco $(j,n+1)$ con un flujo de b_j, si $b_j>0$, para todo nodo $j \in V$. El conjunto L está compuesto por los arcos restantes y el conjunto U es vacío. En este caso, procedemos con el Simplex para redes de la manera habitual, y la solución final será factible (y además óptima), si en la base B, únicamente existe un arco ficticio (en el caso de degeneración habrán varios) con un valor del flujo igual a cero.

Dada la estructura básica, el método Simplex requiere dos pasos básicos: (i) determinar los potenciales de los nodos y (ii) computar los flujos de los arcos.

5.4 Cálculo de los potenciales de los nodos y los flujos de una estructura básica

Consideremos primero el problema de computar los potenciales π para una estructura básica (B,L,U). Asumiremos que $\pi_{n+1} = 0$. Debemos notar que el valor del potencial de un nodo puede ser arbitrario ya que una restricción del problema es redundante. Tenemos $n+1$ incógnitas (tantas como nodos) y n ecuaciones (tantas como arcos en B) Computaremos los restantes potenciales usando las condiciones de optimalidad $\bar{c}_{ij} = 0$ para cada arco (i,j) en B. Estas condiciones se pueden escribir de la forma $\pi_j = \pi_i - c_{ij},\ \forall (i,j) \in \mathrm{B}$.

La idea básica es empezar en el nodo $n+1$ y, usando los índices *Thread*, calcular los otros potenciales. El índice *Thread* permite computar los potenciales de los nodos en tiempo $O(n)$ usando el siguiente método:

```
Procedure Computar_potenciales;
  begin
  π_{n+1} := 0;
  j := Thread(n+1);
  while j ≠ n + 1 do
    begin
    i := Pred(i);
    if (i, j) ∈ A then π_j := π_i − c_ij;
    if (j, i) ∈ A then π_j := π_i + c_ij;
    j := Thread(j)
    end
  end;
```

El siguiente procedimiento, inicializa los flujos de los arcos de la red y, posteriormente, asigna el valor del flujo de los arcos básicos ficticios (ó artificiales). El siguiente pseudo-código explica este procedimiento:

```
Procedure Computar_flujos_iniciales;
  begin
  e_i := b_i;  ∀i ∈ V
  U := ∅;
  for ∀(i, j) ∈ A do
    x_ij := l_ij; e_i:=e_i-l_ij; e_i:=e_i+l_ij; L:= L + (i,j);
  for i:=1 to n do
    begin
    if (e_i ≤ 0) then
      Añade el arco (n+1,j)a B con un flujo de -e_i
    else
       Añade el arco (j,n+1) a B con un flujo de e_i;
    e_{n+1}:=e_i+e_{n+1}; e_i:=0
    end
  end;
```

El procedimiento *Computar_flujos_inciales* requiere un esfuerzo O(m), ya que ha de recorrer todos los arcos.

El pivoteo simplex para redes requiere la determinación del arco entrante, el arco saliente y la actualización de los potenciales y de la estructura básica. A continuación, pasamos a describir estos procedimientos.

5.5 Determinación del arco entrante

Pueden elegirse dos tipos de arcos para entrar en la base: algún arco no básico cuyo flujo esta en su cota inferior con un coste reducido negativo ó algún arco no básico cuyo flujo esta en su cota superior con un coste reducido positivo. Estos arcos violan las condiciones de optimalidad. El criterio utilizado para la selección de un arco entrante entre los elegibles tiene un efecto importante en la eficiencia del algoritmo Simplex. Una implementación que selecciona un arco que viola las condiciones de optimalidad, es la de elegir aquel arco que tiene el mayor valor $|\bar{c}_{ij}|$ de todos los arcos candidatos (*regla de Dantzig*). Si bien esta selección podría requerir pocas iteraciones en la práctica, examina cada arco en cada iteración, lo que consume mucho tiempo. Por otro lado, examinando una lista de arcos cíclicamente, y seleccionando el primer arco que viola las condiciones de optimalidad, encontraría rápidamente el arco entrante, pero podría requerir muchas iteraciones debido a esta elección pobre (*regla elegir primer arco*). Una de las más exitosas implementaciones usa una técnica de *lista de candidatos* que es un compromiso efectivo entre las dos estrategias anteriores. No obstante, en nuestra implementación del método no hemos decantado por la regla de Dantzig.

5.6 Determinación del arco saliente

Supongamos que seleccionamos el arco (i,j) como arco entrante. La adición de este arco a la base B forma exactamente un ciclo no dirigido W, el cual denominamos *ciclo de pivoteo*. Definimos la orientación de W como la misma que la del arco (i,j), si $(i,j) \in L$ y en dirección opuesta de (i,j), si $(i,j) \in U$. Sea $\vec{W}$ y $\overleftarrow{W}$, respectivamente, los conjuntos de los arcos en W en la misma dirección y en oposición a la orientación del ciclo. Enviando un flujo adicional alrededor del ciclo de pivoteo W en la dirección de su orientación decrece estrictamente el coste de la solución actual. Así, se envía tanto flujo como sea posible hasta que uno de los arcos en el ciclo W alcance su cota inferior ó superior; este arco sale de la base. El máximo cambio de flujo δ_{kl} sobre un arco $(K,l) \in W$ es:

$$\delta_{kl} = \begin{cases} u_{kl} - x_{kl} \ \ si\ (i,j) \in \overrightarrow{W} \\ x_{kl} - l_{kl} \ \ si\ (i,j) \in \overleftarrow{W} \end{cases}$$

Enviamos $\delta = \min\{\delta_{kl} : (k,l) \in W\}$ unidades de flujo alrededor de *W*, y seleccionamos un arco (*p,q*) con $\delta_{pq} = \delta$ como el arco saliente. La operación crucial es identificar el ciclo *W*. Usando el índice *Pred* identificaremos el ciclo *W* como sigue: Empezando en el nodo *i* y usando el índice predecesor, trazamos el camino desde este nodo a la raíz. Repetimos la misma operación empezando en el nodo *j* hasta encontrar un nodo *w*, el cual es el primer ascendiente común de los nodos *i* y *j*. Se puede hacer esta operación simultáneamente para los nodos *i* y *j* usando los índices *Pred* y *Depth*, de la manera siguiente:

```
Procedure Identificar_ciclo;
  begin
  k := i ;
  l := j ;
  while k ≠ l do
    begin
    if Depth(k) > Depth(l) then k := Pred(k)
    else
      if Depth(k) < Depth(l) then l := Pred(l)
      else
        begin
        k := Pred(k);
        l := Pred(l)
        end
    end;
  w := k
  end;
```

Una simple modificación de este procedimiento permite identificar el flujo δ que puede ser enviado a lo largo de *W*, así como determinar el primer ascendiente común *w* de *i* y *j*. Esta operación requiere un esfuerzo O(*n*), en el caso peor.

5.7 Actualización de la estructura básica y de los potenciales

En la terminología del método Simplex, un cambio de base es una operación de pivoteo. Si $\delta = 0$, entonces se dice que el pivoteo es degenerado; en otro caso es no degenerado. Una base se dice

degenerada si el flujo de algún arco básico es igual a su cota superior ó inferior y no degenerado en otro caso. Obsérvese que un pivoteo degenerado ocurre sólo en bases degeneradas.

Cada vez que entra un arco (*i,j*) y sale un arco (*p,q*), se debe actualizar la estructura básica. Si el arco saliente es el mismo que el arco entrante, lo que ocurre cuando $\delta = u_{ij} - l_{ij}$, la base no cambia. En este caso, el arco (*i,j*) se mueve del conjunto *L* al conjunto *U*, ó al revés. Si el arco saliente difiere del arco entrante, se necesitan hacer más cambios. En este caso, el arco (*p,q*) pasa a ser un arco no básico en su cota inferior ó superior dependiendo de si $x_{pq} = l_{pq}$ ó $x_{pq} = u_{pq}$. Añadiendo (*i,j*) a la base y borrando (*p,q*) de la base, obtenemos una nueva. Los potenciales de los nodos cambian y pueden ser actualizados como sigue. La salida del arco (*p,q*) de la base divide el conjunto de nodos en dos subárboles: T_1 , que contiene el nodo raíz y T_2 que no contiene la raíz. Hay que notar que el subárbol T_2 cuelga del nodo *p* o del nodo *q*. El arco (*i,j*) tiene un nodo T_1 y el otro en T_2. Como $\pi_{n+1} = 0$ y $c_{kl} - \pi_k + \pi_l = 0$ para todos los arcos en la nueva base, se tiene que los potenciales de los nodos en T_1 no cambian y los potenciales de los nodos en T_2 deben cambiar por una constante. Si $i \in T_1$ y $j \in T_2$, entonces los potenciales de los nodos en T_2 disminuyen en $-\bar{c}_{ij}$; si $j \in T_1$ y $i \in T_2$, aumentan en $\bar{c}_{ij}$. El siguiente método, usando los índices *Thread* y *Depth*, actualiza los potenciales rápidamente:

```
Procedure Actualizar_potenciales;
  begin
  if q ∈ T2 then y := q
  else y := p;
  if i ∈ T1 then cambio := -c̄ij
  else cambio := c̄ij;
  πy ≔ πy + cambio;
  z := Thread(y);
  while Depth(z) < Depth(y) do
    begin
    πz ≔ πz + cambio;
    z := Thread(z)
    end
  end
```

El último paso en los cambios de la base es actualizar los índices. Es posible actualizar los 3 índices en O(n) con un recorrido en profundidad de la base (DFS).

5.8 Terminación del algoritmo

El algoritmo Simplex para redes se mueve desde una estructura básica a otra hasta obtener una estructura básica que satisface las condiciones de optimalidad. Se puede mostrar fácilmente que el algoritmo termina en un número finito de pasos si cada operación de pivoteo es no degenerada. Debemos recordar que $\left|\bar{c}_{ij}\right|$ representa la disminución neta en el coste por unidad de flujo enviado alrededor del ciclo W. Durante un pivoteo no degenerado, la nueva estructura básica tiene un coste de $\delta \cdot \left|\bar{c}_{ij}\right|$ unidades menor que la estructura básica previa. En virtud de que hay un número finito de estructuras básicas y que toda estructura básica tiene asociado un único coste, el algoritmo Simplex terminará en un número finito de pasos si no hay degeneración.

5.9 Complejidad del Algoritmo.

El algoritmo Simplex para redes podría no terminar necesariamente en un número finito de iteraciones, salvo que impongamos una restricción adicional sobre la elección de los arcos entrantes y salientes. Esto es importante para evitar ciclados, es decir, una secuencia repetitiva infinita de pivoteos degenerados. Veremos que manteniendo un tipo especial de bases, llamadas *bases fuertemente factibles*, el algoritmo Simplex termina finalmente, más aún, va más rápido también en la práctica.

Sea (B,L,U) una estructura básica del problema de flujo de coste mínimo con datos enteros. Los arcos del árbol, ó están apuntando hacia arriba (hacia la raíz) ó están apuntando hacia abajo. Diremos que una estructura básica (B,L,U) es *fuertemente factible* si se puede enviar una cantidad de flujo positivo desde cualquier nodo del árbol a la raíz a lo largo de arcos del árbol sin violar alguna de las cotas de sus capacidades.

La *técnica de perturbación* es un método conocido para evitar el ciclado (degeneración) en el algoritmo Simplex Primal. La técnica perturba el vector del lado derecho (oferta/demanda) de tal manera que toda base factible sea no degenerada y una solución óptima del problema perturbado pueda ser convertida fácilmente a una solución óptima del problema original.

El problema de flujo de coste mínimo puede ser perturbado cambiando el vector de oferta/demanda de b a $b+\varepsilon$. Diremos que $\varepsilon = (\varepsilon_1, \varepsilon_2, ..., \varepsilon_n)$ es una perturbación factible si satisface las condiciones siguientes:

$$\text{a) } \varepsilon_i > 0,\ \forall i = 2,...,n;\ \text{ b) } \sum_{i=2}^{n} \varepsilon_i < 1;\ \text{ c) } \varepsilon_1 = -\sum_{i=2}^{n} \varepsilon_i;$$

Una posible elección para una perturbación es $\varepsilon_i = 1/n,\ i = 2,...,n,$ y así $\varepsilon_1 = -(n-1)/n$. Otra elección es $\varepsilon_i = \alpha^i,\ i = 2,...,n,$ donde α es un número positivo muy pequeño. Con respecto a una estructura básica, la perturbación cambia el flujo sobre los arcos básicos. Si (i,j) es un arco apuntando hacia abajo en el árbol B y $D(j)$ son los descendientes del nodo j, entonces la perturbación hace decrecer el flujo en el arco (i,j) por $\sum_{k \in D(j)} \varepsilon_k$. Si (i,j) es un arco apuntando hacia arriba en el árbol B y $D(i)$ son los descendientes del nodo i, entonces la perturbación incrementa el flujo en el arco (i,j) por $\sum_{k \in D(i)} \varepsilon_k$. Hay que notar que ambas cantidades de flujo son no enteras y distintas de cero y, por lo tanto, todo pivoteo es no degenerado y el arco que sale de la base es único. Esto lleva al siguiente resultado:

Teorema 1.5. *Para alguna estructura básica (B,L,U) del problema de flujo de coste mínimo, las siguientes afirmaciones son equivalentes:*

a) (B,L,U) *es fuertemente factible.*

b) (B,L,U) *es factible si* b *es reemplazado por* $b+\varepsilon$ *para alguna perturbación factible* ε.

c) *Ningún arco apuntando hacia arriba en la base B tiene un flujo igual a su cota superior y ningún arco apuntando hacia a bajo en la base B tiene un flujo igual a su cota inferior.*

Este teorema demuestra que mantener una base fuertemente factible es equivalente a aplicar el algoritmo Simplex al problema perturbado. Este resultado indica que ambas herramientas obtienen exactamente la misma secuencia de estructuras básicas si ambas usan la misma regla para seleccionar los arcos entrantes. Como un corolario, esta equivalencia muestra que cualquier implementación del algoritmo simplex que mantenga una base fuertemente factible realiza como mucho $nmCU$ pivoteos. Para ver este resultado, consideremos el problema perturbado con la perturbación $(-(n-1/n),1/n,1/n,....,1/n)$. Con esta perturbación, el flujo sobre todo arco es un múltiplo de $1/n$ y así, el valor de la función objetivo decrece al menos por $1/n$ unidades. Ya que mCU es una cota superior del valor de la función objetivo de la solución al comienzo y cero es el valor de la cota inferior, el algoritmo terminará como mucho en $nmCU$ iteraciones. De esta manera, cualquier implementación del algoritmo simplex que mantiene una base fuertemente factible, requiere un tiempo pseudopolinomial.

Hemos visto que con este tipo de bases realizamos $O(nmCU)$ pivoteos. Usando la regla de pivoteo de Dantzig, es decir, la de considerar como arco entrante aquel cuyo $|\bar{c}_{ij}|$ tiene el mayor valor y viola las condiciones de optimalidad, podemos reducir el número de pivoteos a $O(nmU\log H)$, siendo $H=mCU$. Este resultado es dado en el siguiente teorema.

Teorema 1.6. *El algoritmo Simplex que mantiene una base fuertemente factible y usa la regla de Dantzig realiza $O(nmU\log H)$ pivoteos.*

6. Problema de flujo máximo

Como caso particular del problema de flujo de coste mínimo, el problema de flujo máximo se puede introducir de la forma siguiente. Dada una red dirigida, $G=(V,A)$, sea $V=\{1,..,n\}$ el conjunto

de n nodos y A el conjunto de m arcos. Distinguimos dos nodos especiales en G: el nodo fuente s y el nodo sumidero t. Dado cualquier nodo i, definimos el conjunto $\text{Pred}(i) = \{j \in V / (j,i) \in A\}$ y el conjunto $\text{Suc}(i) = \{j \in V / (i,j) \in A\}$. Cada arco $(i,j) \in A$ tiene asociado los siguientes valores: la capacidad u_{ij} que denota la mayor cantidad de flujo que puede soportar el arco (i,j), y l_{ij}, la correspondiente cota inferior del flujo sobre el arco. Sin pérdida de generalidad podemos suponer que l_{ij} es igual a cero para cada arco (i,j) (ver sección (3.1.2)). Definimos la lista de arcos que salen del nodo i al conjunto $\text{A}(i) = \{(i,j) \in A : j \in Suc(i)\}$. Denotaremos por U al máximo valor del conjunto $\{u_{ij} / (i,j) \in A\}$. Un *flujo* es una función $x : A \to R^+ \cup \{0\}$ que satisface:

$$\sum_{j \in Suc(i)} x_{ij} - \sum_{j \in Pred(i)} x_{ji} = \begin{cases} f & \text{si } i = s \\ 0 & i \in V - \{s,t\} \\ -f & \text{si } i = t \end{cases} \qquad (1.2a)$$

$$0 \le x_{ij} \le u_{ij}, (i,j) \in A \qquad (1.2b)$$

para algún $f \geq 0$. El problema de flujo máximo consiste en encontrar un flujo x tal que f es maximizado.

Se define la *capacidad de un s-t corte* mediante la suma de las capacidades de los arcos que salen del conjunto S y llegan a $\overline{S}$, esta capacidad la denotamos por $u[S,\overline{S}] = \sum_{(i,j) \in [S,\overline{S}]} u_{ij}$.

Un *camino incremental* en la red residual R es un camino dirigido de s a t en R. Definimos la *capacidad residual* de un camino como la menor de las capacidades residuales de los arcos en el camino.

Teorema 1.7. (*Teorema del Flujo-Máximo Corte-Mínimo*) *Si f es el valor del flujo x en la red G=(V,A) con la fuente s y el sumidero t, entonces las siguientes condiciones son equivalentes:*

1) *f es el valor del flujo máximo en G*
2) *La red residual R=G(x) no contiene caminos incrementales.*

3) $f = U[S, \overline{S}]$ *para algún s-t corte de G.*

La condición 2) describe un criterio de parada que es utilizado para obtener el algoritmo conocido como *algoritmo de Caminos Incrementales* dado por Ford y Fulkerson que a continuación procedemos a introducir.

6.1 Algoritmo de Ford y Fulkerson

Este primer algoritmo considera en cada iteración un camino incremental en *R* y envía a través de él una cantidad de flujo igual a la capacidad residual del camino. El algoritmo termina cuando no quedan caminos incrementales en *R*. El algoritmo genérico de caminos incrementales es de la manera siguiente:

```
Algoritmo de Ford-Fulkerson;
  begin
  x := 0; F:=0;
  while haya un camino P de s a t en R do
    begin
    Δ := min {r_ij : (i, j) ∈ P}; {Δ es el cuello de botella del
    camino}
    Enviar Δ unidades de flujo a lo largo de P y actualizar R;
    F := F + Δ
    end
  end.
```

Para cada arco $(i, j) \in P$ actualizamos $r_{ij} = r_{ij} - \Delta$ y $r_{ji} = r_{ji} + \Delta$. Supongamos que actualizamos las capacidades residuales en algún punto del algoritmo y que nos preguntamos por el efecto sobre los flujos de los arcos. De la definición de la capacidad residual tenemos que un flujo adicional de Δ unidades sobre el arco (i, j) en la red residual *R*, corresponde a: (1) un incremento de x_{ij} en Δ unidades en la red original, ó (2) un decremento de x_{ji} en Δ unidades en la red original, ó (3) una combinación de ambas.

Finalmente, supongamos que están dados los valores para las capacidades residuales. ¿Cómo podríamos determinar los flujos x_{ij}? Observamos que $r_{ij} = u_{ij} - x_{ij}$ y $r_{ji} = x_{ij}$ y, por lo tanto, podemos obtener los valores de x_{ij} haciendo $x_{ij} = u_{ij} - r_{ij}$ ó $x_{ij} = r_{ji}$.

En la descripción del algoritmo precedente, no se discutieron algunos detalles importantes: (1) cómo identificar un camino incremental o mostrar que la red no contiene tales caminos, y (2) si el algoritmo termina en un número finito de iteraciones y si, cuando lo hace, obtiene un flujo máximo.

Introduciremos a continuación una implementación específica del algoritmo genérico de caminos incrementales conocido como *algoritmo de etiquetado*. El algoritmo de etiquetado usa una técnica de búsqueda para identificar un camino en R de la fuente al sumidero. El algoritmo parte del nodo fuente encontrando a todos los nodos que son alcanzables desde la fuente a lo largo de un camino en la red residual. En cualquier paso, el algoritmo divide el conjunto de nodos de la red en dos grupos: etiquetados y no etiquetados. Los nodos etiquetados son aquellos nodos que el algoritmo ha alcanzado en el proceso de búsqueda; los nodos no etiquetados son aquellos nodos que el algoritmo todavía no ha alcanzado en el proceso. Cuando el nodo sumidero es etiquetado, el algoritmo envía la máxima cantidad de flujo posible sobre el camino desde s a t. En este punto, se borran las etiquetas y se repite el proceso. El algoritmo termina cuando, examinados todos los nodos etiquetados, el nodo sumidero permanece sin etiquetar, implicando que el nodo fuente no está conectado al nodo sumidero en la red residual.

El siguiente esquema especifica los pasos de etiquetado. Se necesita un índice $Pred(i)$, para cada nodo etiquetado i, que indica el nodo que motivó la asignación de etiqueta a i. Este índice será utilizado posteriormente para indicar el camino de s a t, sin más que empezar con $i = t$ y, con $Pred(i)$, llegar hasta s.

```
Procedure Aumentar;
  begin
  Utiliza los índices de los predecesores para identificar un
  camino incremental P de la fuente al sumidero;
  δ := min{r_ij : (i, j) ∈ P}; F := F + δ;
  Envía δ unidades de flujo a lo largo de P y actualiza las
  capacidades residuales
  end;
```

```
Algoritmo de Etiquetado;
  begin
  x:=0; F:=0;
  repeat
    Etiquetado[j]:= false y Pred(j):=0 para todos los nodos;
    Etiquetado[s]:= true; L := {s};
    while (L ≠ ∅) and not(etiquetado[t]) do
      begin
      selecciona un nodo i ∈ L;
      for (i,j) ∈ A(i) do
        if j no esta etiquetado y r_ij > 0 then
          begin
          Pred(j):= i;
          Etiquetado[j] := true;
          L := L + {j}
          end
      end;
    if etiquetado[t] then Aumentar
  until not(etiquetado[t]);
  end;
```

El algoritmo de etiquetado anterior termina obteniendo un flujo máximo, debido a que su criterio de parada es la no existencia de caminos incrementales. Ahora veremos en cuántos pasos determina el flujo máximo basándonos en el criterio del caso peor. En cada iteración del algoritmo, se etiqueta algún nodo i al menos una vez, inspeccionando cada arco de $A(i)$. Por lo tanto, en el proceso de etiquetado, se examina cada arco al menos una vez. Esto requiere $O(m)$ computaciones. Si todas las capacidades de los arcos son enteras y acotadas por un número finito U, entonces la capacidad del s-t corte dado por $(s, V\text{-}\{s\})$ es a lo sumo nU. Este nos da una cota superior del valor del flujo máximo. Como el algoritmo de etiquetado incrementa el valor del flujo en al menos una unidad en cada iteración, necesita entonces nU iteraciones para obtener dicho valor de flujo. Esto implica que el algoritmo tiene una complejidad $O(nmU)$.

El algoritmo de etiquetado es posiblemente el algoritmo más simple para la resolución del problema de flujo máximo. En la práctica, el algoritmo trabaja razonablemente bien. Sin embargo, la cota del caso peor sobre el número de iteraciones no es enteramente satisfactoria para valores de U grandes. Por ejemplo, si $U = 2^n$, la cota sería exponencial en el número de nodos. Más aún, el algoritmo puede de hecho realizar muchas iteraciones como en el

ejemplo dado en la Figura 1.15. En esta red, el algoritmo puede seleccionar los caminos incrementales *s-a-b-t* y *s-b-a-t* alternativamente 10^6 veces, enviando cada vez una unidad de flujo a lo largo del correspondiente camino incremental.

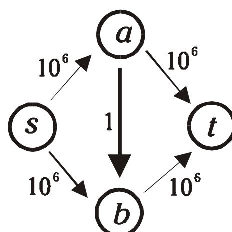


Figura 1.15. Ejemplo patológico del algoritmo de etiquetado.

Una segunda desventaja del algoritmo de etiquetado es que si las capacidades son irracionales, el algoritmo podría no terminar. Para algunos casos patológicos del problema de flujo máximo, el algoritmo de etiquetado no termina, y aunque los sucesivos valores de flujo convergen, estos convergerán a un valor estrictamente menor que el valor del flujo máximo. Por lo tanto, si queremos garantizar la efectividad del algoritmo, debemos seleccionar los caminos incrementales cuidadosamente. Una tercera desventaja del algoritmo de etiquetado es su “olvido”. En cada iteración, el algoritmo genera etiquetas de los nodos que contienen información acerca de los caminos incrementales desde la fuente a otros nodos. La implementación descrita borra las etiquetas cada vez que se pasa de una iteración a la siguiente, si bien mucha de esta información podría ser válida en la siguiente iteración. Eliminando las etiquetas, por lo tanto se destruye información potencial. Idealmente, retener las etiquetas puede ser provechoso en futuros cálculos.

6.2 Algoritmos de caminos incrementales mínimos

Edmonds y Karp [26] demuestran que si en cada iteración del algoritmo de Ford y Fulkerson se selecciona el camino incremental mínimo, se necesitan como máximo $mn/2$ iteraciones, obteniendo así un algoritmo de complejidad fuertemente polinomial. Para la obtención de los caminos mínimos se utiliza un recorrido en

amplitud en el grafo que requiere un esfuerzo de O(m). Así se obtiene un algoritmo de complejidad O(m^2n).

Un algoritmo de caminos incrementales mínimos, que utiliza las etiquetas distancias, es debido a Ahuja y Orlin [5] y tiene una complejidad O(mn^2). Discutiremos en detalle este algoritmo debido a que será utilizado en el capítulo 2. Este algoritmo utiliza las etiquetas distancias introducidas por Goldberg [38]. Sea una *función distancia* $d: V \rightarrow Z^+ \cup \{0\}$ con respecto a las capacidades residuales r_{ij}. Se dice que es una función distancia *válida* si satisface las siguientes dos condiciones:

$$a)\ d(t) = 0$$
$$b)\ d(i) \le d(j) + 1,\ \forall (i,j) \in A, r_{ij} > 0$$

Llamaremos *etiqueta distancia* de un nodo i a $d(i)$. Mediante inducción, se puede demostrar que $d(i)$ es una cota inferior de la longitud del camino mínimo desde el nodo i al nodo t en R, donde la longitud del camino viene dada por el número de arcos que este utiliza. Si para cada nodo i, $d(i)$ coincide con la longitud del camino mínimo desde el nodo i al nodo t en R, entonces diremos que las etiquetas distancias son *exactas*. Si $d(s) \ge n$, entonces no hay camino incremental en R.

Un arco (i,j) en la red residual se denomina *admisible* si satisface la siguiente condición: $d(i)=d(j)+1$. Un camino incremental en R que consiste de arcos admisibles es denominado *camino admisible*. Un camino admisible es un camino incremental mínimo de s a t. Dicho algoritmo tiene el siguiente esquema:

```
Procedure Avanzar(i);
  begin
  Sea (i,j) un arco admisible que sale de i;
  pred(j) := i; i := j;
  end;

Procedure Retroceder(i);
  begin
  d(i) := min{d(j) + 1 : (i, j) ∈ A(i) y r_ij > 0};
  if i ≠ s then i := pred(i)
  end;
```

```
Algoritmo Caminos_Incrementales_Mínimos;
  begin
  X := 0; d(t):= 0;
  Obtener las etiquetas distancias exactas mediante BFS inverso
  en la red residual empezando en el nodo sumidero t;
  i := s;
  while d(s)< n do
    if i tiene un arco admisible then
      begin
      Avanzar(i);
      if i = t then
        begin
        Aumentar;
        i = s
        end
      end
    else
      Retroceder(i)
  end.
```

Este algoritmo de caminos mínimos incrementales procede enviando flujos a lo largo de caminos admisibles. El algoritmo realiza primero un recorrido en amplitud inverso en R empezando desde el nodo sumidero t para computar las etiquetas distancias exactas $d(i)$. El algoritmo mantiene un índice de predecesor con cada nodo de la red, de tal manera que $pred(i)$ almacena el nodo anterior al nodo i en el actual camino admisible de s a i. El algoritmo realiza iterativamente pasos *Avanzar* o *Retroceder* sobre el último nodo del camino admisible parcial. Si del nodo actual i sale un arco admisible (i, j), entonces realiza un paso *Avanzar* y añade este arco al camino parcial actual. Si esto no ocurre, realiza un paso *Retroceder* que incrementa la etiqueta distancia del nodo i, haciendo que el arco $(pred(i),i)$ sea no admisible y, por lo tanto, retrocediendo un arco sobre el camino parcial. Si en este proceso se alcanza al nodo sumidero, se envía una cantidad de flujo igual a la capacidad residual del camino admisible incremental. Esta operación la realiza el procedimiento *Aumentar*. El proceso se repite hasta que la etiqueta distancia del nodo fuente s sea mayor o igual que n, lo que significa que no hay camino incremental en R y, por lo tanto, el flujo es máximo.

Hemos de comentar que el algoritmo de caminos incrementales mínimos utiliza la siguiente estructura de datos para identificar arcos admisibles que salgan de un nodo. Cada nodo i tiene un arco

que pertenece a $A(i)$ que es el actual candidato para examinar si es admisible; a este arco lo denominaremos arco actual de i. Al principio el arco actual de i es el primer arco de $A(i)$. Si el algoritmo alcanza al nodo i, se examina si este arco es admisible. Si el arco actual de i no es admisible, entonces el arco actual de i pasa a ser el siguiente elemento de la lista de $A(i)$. El proceso se repite hasta que se encuentre un arco admisible o hasta alcanzar el final de la lista de arcos. Si ocurre esto último, significa que $A(i)$ no contiene ningún arco admisible y, por lo tanto, se realiza un paso de *Retroceder*(i) y el arco actual de i pasa a ser de nuevo el primer elemento de $A(i)$.

El algoritmo de caminos incrementales mínimos termina cuando $d(s) \geq n$, indicando que la red no contiene caminos incrementales de la fuente al sumidero. Consecuentemente, el flujo obtenido en la terminación del algoritmo es un flujo máximo.

A continuación mostraremos que la complejidad del algoritmo de caminos incrementales mínimos es $O(mn^2)$.

Propiedad 1.1. Si el algoritmo actualiza la distancia (reetiqueta) de cualquier nodo a lo sumo k veces, el esfuerzo empleado en determinar arcos admisibles y en las llamadas a retroceder es $O(k\sum_{i \in V}|A(i)|)=O(km)$.

Se dice que un arco $(i,j) \in A$ es *saturado* si, tras enviar flujo sobre él, su capacidad residual se hace cero.

Lema 1.1. *Si el algoritmo reetiqueta cualquier nodo a lo sumo k veces, entonces satura los arcos a lo sumo $km/2$ veces.*

Demostración. Mostraremos que entre dos saturaciones consecutivas de un arco (i,j), ambas etiquetas $d(i)$ y $d(j)$ deben incrementar en al menos dos unidades. Ya que, por hipótesis, el algoritmo incrementa cada etiqueta distancia a lo sumo k veces, este resultado implicaría que el algoritmo podría saturar cualquier arco a lo sumo $k/2$ veces. Por tanto, el número total en el caso peor de saturaciones de todos los arcos sería $km/2$.

Supongamos que un envío satura un arco (i,j). Ya que el arco (i,j) es admisible, $d(i)=d(j)+1$. Antes que el algoritmo sature este arco otra vez, debe enviar flujo de vuelta desde el nodo j al nodo i. En este instante, las etiquetas distancia $d'(i)$ y $d'(j)$ satisfacen la igualdad $d'(j)=d'(i)+1$. En la siguiente saturación del arco (i,j), debemos tener $d''(i)=d''(j)+1$. Así se tiene que $d''(i)=d''(j)+1\geq d'(j)+1=d'(i)+2\geq d(i)+2$, similarmente es posible mostrar que $d''(j)\geq d(j)+2$. De esta manera, se demuestra que entre dos saturaciones consecutivas del arco (i,j), ambos $d(i)$ y $d(j)$ incrementan en al menos 2 unidades. □

Lema 1.2.

a) *En el algoritmo de caminos incrementales mínimos, cada etiqueta distancia es incrementada a lo sumo n veces. De esta manera el número total de operaciones Retroceder es a lo sumo,* n^2.

b) *El número de operaciones Aumentar está acotado superiormente por* $nm/2$.

Demostración. Cada operación Retroceder sobre un nodo i incrementa el valor de $d(i)$ en al menos una unidad. Después que el algoritmo ha reetiquetado el nodo i a lo sumo n veces, $d(i)\geq n$. En este punto, el algoritmo nunca selecciona el nodo i durante una operación avanzar ya que, para todo nodo k en el camino admisible parcial, se cumple que $d(k)<d(s)<n$. Así, el algoritmo reetiqueta a un nodo a lo sumo n veces y el número de operaciones retroceder está acotado por n^2. Por el lema 1.1 y el resultado precedente se tiene que el algoritmo realiza a lo sumo $nm/2$ saturaciones. Ya que cada operación Aumentar satura al menos un arco, el número de operaciones Aumentar esta acotado por $nm/2$. □

Teorema 1.8. *El algoritmo de caminos incrementales mínimos requiere un tiempo* $O(n^2m)$.

Demostración. Usando los resultados anteriores, encontramos que el esfuerzo empleado en encontrar arcos admisibles y reetiquetar los nodos es O(nm). El lema 1.2 implica que el número total de operaciones *Aumentar* es O(nm). Ya que cada operación Aumentar requiere un tiempo O(n), el esfuerzo total de estas operaciones es O(n^2m). Cada operación *Retroceder* reetiqueta un nodo; así, el número total de operaciones *Retroceder* es O(n^2). Cada operación *Avanzar* añade un arco al camino admisible parcial y cada operación *Retroceder* elimina un arco del mismo. Ya que cada camino admisible parcial tiene una longitud de, a lo sumo, n, el algoritmo requiere O($n^2 + n^2m$) operaciones *Avanzar*. El primer término se refiere al número de operaciones *Retroceder* y el segundo se refiere al número de envíos de flujo (*Aumentar*). La combinación de estas cotas establece el teorema. □

Los precedentes lemas y teoremas nos serán de utilidad en el capítulo 2 y 3 de la presente memoria.

A continuación, si entrar en excesivos detalles comentaremos algunas ideas que son utilizadas en los algoritmos de flujo máximo propuestos en fechas más recientes.

6.3 Escalado en las capacidades y preflujo

Como hemos dicho previamente, Edmonds y Karp [26], proponen una modificación del algoritmo de Ford y Fulkerson que, en cada iteración, selecciona el camino incremental de mayor cuello de botella. Para identificar dicho camino proponían una modificación del algoritmo de Dijkstra. El algoritmo realiza un número de iteraciones O($nm\log U$) y tiene una complejidad O($m^2n\log U$).

Una variación del anterior algoritmo es el *algoritmo de escalado en las capacidades* dado por Ahuja y Orlin [5]. La idea esencial subyacente en este nuevo algoritmo es conceptualmente bastante simple: Enviar flujo a lo largo de un camino incremental con capacidad residual suficientemente grande, en lugar del camino de máxima capacidad, debido a que podemos obtener un camino con

una capacidad residual suficientemente grande fácilmente en un tiempo O(m).

Para definir el escalado en las capacidades introducimos un parámetro Δ y nos referimos a la red *Δ-residual*, $R(\Delta)$, como a la red que únicamente contiene arcos cuya capacidad residual es al menos Δ. De esta manera, todos los caminos incrementales en $R(\Delta)$ tienen una capacidad residual de al menos Δ. Se dice que un flujo es *Δ-óptimo* si en R no existe ningún camino incremental con capacidad residual de al menos Δ.

El algoritmo trabaja en fases de escalado, cada fase tiene un Δ fijo y, cuando finaliza una fase, comienza la siguiente con $\Delta=\Delta/2$. Al principio $\Delta = 2^{\lfloor \log U \rfloor + 1}$, es decir, es igual al menor entero potencia de dos que es mayor o igual que U. La última fase del algoritmo ocurre cuando $\Delta=1$. En esta última fase $R(\Delta)=R$ y, por tanto, al finalizar el algoritmo se determina el flujo máximo. De esta manera, el algoritmo realiza $\lfloor \log U \rfloor + 1$ fases de escalado. El esquema del algoritmo es como sigue:

```
Algoritmo Escalado_en_la_capacidad;
  begin
  X:=0;
  Δ := 2^⌈log U⌉;
  while Δ ≥ 1 do
    begin
    While Exista una camino P de s a t en R(Δ) do
      begin
      Identifica un camino P de s a t;
      δ := min{r_ij : (i, j) ∈ P};
      Envía δ unidades de flujo a lo largo de P y actualiza R(Δ)
      end;
    Δ := Δ / 2
    end
  end
```

La eficiencia del algoritmo depende del hecho de que realiza a lo sumo $2m$ envíos de flujo por fase de escalado. Para ver este resultado, consideremos el flujo al final de la fase Δ-escalado. Sea x' el flujo factible y f' el valor del flujo. Además sea S, el conjunto de nodos alcanzables desde s en $R(\Delta)$, ya que $R(\Delta)$ no contiene un camino incremental desde la fuente al sumidero $t \notin S$. De esta manera, $[S, \bar{S}]$ forma un s-t corte. La definición de S implica que la

capacidad residual del corte $[S,\overline{S}]$ es a lo sumo $m\Delta$. De esta manera $f^* - f' \leq m\Delta$. En la próxima fase de escalado, cada incremento es de al menos $\Delta/2$ unidades de flujo. Así, en esta fase, se realizan a lo sumo $2m$ incrementos en el flujo.

El algoritmo de etiquetado descrito anteriormente requiere un esfuerzo O(m) para identificar un camino incremental, y actualizar la red $R(\Delta)$ requiere también un tiempo O(m). Dicho de otro modo, el algoritmo de escalado en capacidad resuelve el problema de flujo máximo con O($m\log U$) incrementos de flujo y requiere un tiempo O($m^2 \log U$).

Podemos mejorar el tiempo de ejecución del algoritmo usando el algoritmo de caminos incrementales mínimos como subrutina. Podemos lograr esto definiendo las etiquetas distancia con respecto a la red $R(\Delta)$ e incrementando el flujo a lo largo de caminos admisibles en dicha red. Así, obtendríamos un algoritmo de complejidad O($nm\log U$).

Hasta ahora únicamente hemos considerado algoritmos de caminos incrementales, es decir, algoritmos que en cada iteración identifican un camino incremental en la correspondiente red residual. La inherente desventaja de los algoritmos de caminos incrementales es la computacionalmente costosa operación de envío de flujo a lo largo de un camino, la cual requiere un tiempo O(n) en el caso peor. A los algoritmos de *preflujo* no les afecta esta desventaja, obteniendo mejoras respecto a la complejidad del caso peor. Esto es así porque que los algoritmos de preflujo envían flujo sobre arcos individuales en lugar de a lo largo de caminos incrementales.

Debido a esta última particularidad, los algoritmos de preflujo no satisfacen las restricciones de conservación de flujo en pasos intermedios. De hecho, estos algoritmos permiten que el flujo entrante en un nodo exceda del flujo saliendo del nodo. Llamaremos a tal solución un *preflujo*. Más formalmente, un preflujo es una función $x: A \rightarrow R$ que satisface las restricciones de acotación de los arcos y la siguiente relajación de la restricción de conservación:

$$\sum_{j \in Pred(i)} x_{ji} - \sum_{j \in Suc(i)} x_{ij} \geq 0 \quad \forall i \in V - \{s,t\}$$

La idea de preflujo fue introducida por Karzanov en 1974 [49], a continuación desarrollaremos el algoritmo genérico de preflujo de Golberg [38].

Dado un preflujo x, se define el *exceso* de cada nodo $i \in V$ como:

$$e(i) = \sum_{j \in Pred(i)} x_{ji} - \sum_{j \in Suc(i)} x_{ij} \geq 0$$

En un preflujo, $e(i) \geq 0$ para todo $i \in V - \{s\}$, de esta manera el nodo fuente s es el único nodo con exceso negativo.

Diremos que un nodo con exceso positivo es un *nodo activo* y adoptaremos la convención de que los nodos fuente y sumidero nunca son activos. En los algoritmos de preflujo, la presencia de nodos activos indica que la solución no es factible. En consecuencia, la operación básica en este algoritmo es seleccionar un nodo activo e intentar deshacer su exceso enviando flujo a sus vecinos. Pero ya que queremos enviar flujo al sumidero, enviamos flujo a los nodos más cercanos al sumidero. Mediremos la cercanía al sumidero utilizando las etiquetas distancias. Así, enviar flujo próximo al sumidero es equivalente a enviar flujo sobre arcos admisibles. Luego, el flujo únicamente se envía sobre arcos admisibles. Si el nodo activo seleccionado en una iteración no posee arcos admisibles, incrementamos su etiqueta distancia hasta que posea al menos un arco admisible. El algoritmo termina cuando la red no contiene un nodo activo. Los procedimientos del algoritmo genérico de preflujo son los siguientes:

```
Procedure Prepoceso;
  begin
  x_sj = u_sj ∀(s, j) ∈ A ; x_ij = 0 para el resto de arcos; d(t):= 0;
  Obtener las etiquetas distancias exactas mediante un BFS
  inverso en la red residual empezando en el nodo t;
  d(s) := n
  end;
```

```
Procedure Enviar(i);
  begin
  Sea (i,j) un arco admisible que sale de i;
  Envía Δ = min{e(i), r_ij} unidades de flujo desde el nodo i al j
  end;
```

```
Procedure Reetiquetar(i);
  begin
  d(i) := min{d(j) + 1 : (i, j) ∈ A(i) y r_ij > 0}
  end;
```

Un envío de Δ unidades de flujo desde el nodo i al nodo j hace decrecer a $e(i)$ y r_{ij} en Δ unidades e incrementa $e(j)$ y r_{ji} en Δ unidades. Diremos que un envío de Δ unidades de flujo sobre un arco (i,j) es saturante si $\Delta = r_{ij}$ y no saturante en otro caso. Un envío no saturante en un nodo i reduce $e(i)$ a cero. Llamaremos al proceso de incrementar la etiqueta distancia de un nodo, operación de reetiquetado. La versión del algoritmo genérico de preflujo combina los procedimientos descritos anteriormente:

```
Algoritmo Preflujo;
  begin
  Preproceso;
  while exista un nodo activo ≠ {s,t} do
    begin
    selecciona un nodo activo i;
    if i tiene un arco admisible then
      Enviar(i)
    else
      Reetiquetar(i)
    end
  end.
```

La operación de preproceso implica varios aspectos importantes:

a) Causa que todos los nodos adyacentes a s tengan exceso positivo, de tal manera, que tengamos nodos activos para los pasos *Enviar* y *Reetiquetar*.

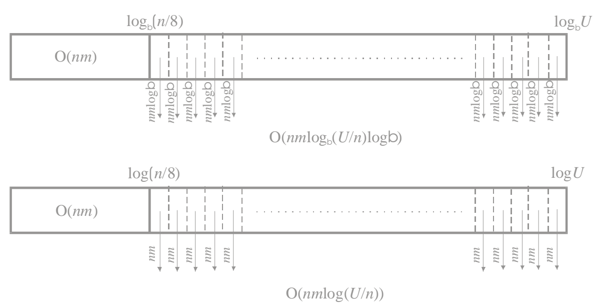
b) Debido a la saturación de los arcos que salen de s, $d(s)$ debe ser igual a n.

c) Como $d(s)=n$, siendo esta una cota inferior del camino mínimo de s a t, implica que no hay camino de s a t en R después del preproceso. Consecuentemente, ya que las etiquetas distancia son no decrecientes, podemos garantizar que, en las siguientes iteraciones, la red

residual nunca contendrá un camino de s a t y, por tanto, no se necesitará enviar flujo desde s otra vez.

Se puede observar que, en cualquier iteración del algoritmo de preflujo, cada nodo i con un exceso positivo esta conectado al nodo s por un camino desde el nodo i al nodo s en la red residual. Esto es importante ya que asegura que, durante una operación de reetiquetado, el algoritmo no minimiza sobre un conjunto vacío.

La complejidad de este algoritmo genérico de preflujo es O(n^2m). Si bien no la demostraremos, si podemos decir que dicha complejidad coincide con el número de envíos no saturantes que realiza el algoritmo en el caso peor. Debemos notar que existe un grado de libertad en el algoritmo genérico en relación con la selección de los nodos activos. Existen diversos criterios de selección de los mismos que conducen a algoritmos con distintas complejidades. Estos criterios serán comentados en un estudio computacional realizado en el siguiente capítulo.



Capítulo 2

Algoritmos para el problema de Flujo Máximo

1. Introducción

Este capítulo está dedicado al análisis del problema de flujo máximo, uno de los problemas de optimización en redes más extensamente estudiado. Este problema fue introducido y formulado con anterioridad en el capítulo 1, en donde también fueron considerados algunos algoritmos importantes.

En la literatura sobre el tema existen diversos estudios computacionales de los algoritmos de flujo máximo. Haremos algunos comentarios sobre los que consideramos más interesantes.

Derigs y Meier [24] realizan una comparación entre distintas versiones del algoritmo de Goldberg y determinan cuáles son los mejores algoritmos para los dos generadores que utilizan.

Ahuja et al.[2] realizan determinadas investigaciones computacionales sobre diez algoritmos de flujo máximo. Estos autores efectúan un estudio empírico usando un recuento de las operaciones representativas que permiten identificar las operaciones cuello de botella de un algoritmo y facilitar la determinación del ratio de crecimiento del mismo. Concluyen que el algoritmo de preflujo genérico con la regla que selecciona el nodo activo con la mayor etiqueta distancia (G_HL), es el mejor en la práctica y que su tiempo de ejecución empírico (virtual) es $O(n^{1.5})$. También suministran una cota empírica para el tiempo de ejecución en la práctica del algoritmo de Dinic. Esta cota es $O(n^{1.6})$.

En este capítulo realizamos un estudio computacional con similares intenciones que el realizado por Ahuja et al.. Nuestro estudio tiene algunas conclusiones en común como es la concordancia sobre el mejor algoritmo en la práctica, pero difiere significativamente en cuanto al diseño del experimento, ya que se consideran más factores.

En Optimización Combinatoria se asume que un buen indicador del comportamiento de un algoritmo es el valor de su complejidad computacional teórica y que la complejidad del caso peor permite clasificar cualquier procedimiento. Sin embargo, algunas veces los

resultados observados en estudios experimentales difieren significativamente de la complejidad teórica, ya que, en algunos problemas, el caso peor rara vez ocurre. Por esta razón, creemos que es necesario complementar la información dada por el indicador previamente mencionado. Para ello se deben considerar otras medidas que nos permitan realizar diferentes clasificaciones de los algoritmos en la práctica. Este hecho ha motivado discusiones acerca de la elección de otros parámetros que permitan realizar esta clasificación práctica. Nuestro interés estriba en estudiar el comportamiento de los algoritmos en la práctica y completar la información dada por la complejidad teórica.

Frecuentemente, los estudios experimentales con diferentes procedimientos incluyen una única lista de los tiempos de CPU obtenidos para un grupo de casos particulares del problema. Además, las comparaciones entre los algoritmos consisten en echar un vistazo a estos valores y así determinar cualquier tipo de orden sobre los algoritmos estudiados. Sin embargo, los casos particulares del problema pueden ser muy valiosos en la clasificación de los procedimientos. Este aspecto es crucial en el desarrollo de este capítulo, el cuál en su primera parte se dedica al estudio de los resultados experimentales de un grupo importante de algoritmos de flujo máximo. Los aspectos fundamentales de este estudio son presentados en el trabajo de Sedeño-Noda et al. [67].

En la segunda parte del capítulo presentamos un algoritmo para resolver el problema de flujo máximo en un tiempo de $O(nm\log(\frac{U}{n}))$ (Sedeño-Noda y González-Martín [68]). Durante las últimas tres décadas los esfuerzos de los investigadores no han culminado en un algoritmo de complejidad $O(nm)$ para cualquier combinación de n y m. Esto es aún cierto cuando $U=\Omega(n)$ (Ahuja et al. [1]). En particular, cuando $m=\Omega(n^2/\log n)$ el algoritmo de Cheriyan et. al. [11] invierte un tiempo $O(nm)$, independientemente del valor de U. Nuestro principal resultado es que, bajo la asunción de que $U=O(n)$, nuestro algoritmo requiere un esfuerzo $O(nm)$ para cualquier valor de n y m. Ningún otro algoritmo alcanza esta complejidad para cualesquiera que sean n y m. Si $U=O(n^K)$ para algún $K>1$, el algoritmo requiere un tiempo $O(nm\log n)$ que coincide con el algoritmo de Sleator y Tarjan [73], sin utilizar estructuras

de datos complejas. El algoritmo que presentamos está basado en el correspondiente de dos fases para redes con capacidades unitarias dado por Ahuja y Orlin [5], incorporando un escalado en las capacidades.

Finalmente, presentamos un algoritmo para resolver el problema de flujo máximo en un tiempo O($nm \log\beta \log_{\beta}(U/n)$) = O($nm \log(U/n)$) para $\beta \leq n/8$ y en un tiempo O($nm \log\beta \log_{\beta} U$) = O($nm \log U$) para $\beta > n/8$. La complejidad del caso peor del algoritmo depende del parámetro β que define la primera escala en las capacidades. Este algoritmo generaliza el algoritmo de dos fases escalado en las capacidades mencionado anteriormente.

Primero Gabow [31], y más tarde Ahuja et al. [5], introducen algoritmos de escala en las capacidades con β=2. En ambos casos, la complejidad teórica es O($nm \log U$). Para estos algoritmos, la complejidad en el caso peor dependiendo de β es O($nm \beta \log_{\beta} U$). Esta cota alcanza un valor mínimo para $\beta = e$. Dado que el número e no es entero, se toma el entero más cercano (2 o 3). El algoritmo propuesto requiere un tiempo O($nm \log\beta \log_{\beta} U$) para $\beta > n/8$. Así, el término β se sustituye por el término $\log\beta$ en la complejidad teórica de los anteriormente mencionados algoritmos.

Estamos convencidos de que el comportamiento en la práctica de estos algoritmos es interesante, debido a que permiten incorporar diferentes estrategias para mejorar el tiempo de CPU y considerar diferentes valores de β. En la parte final de este capítulo realizamos un análisis experimental de estos algoritmos obteniendo el tiempo de CPU virtual para los mismos.

2. Estudio algorítmico del problema de flujo máximo: un análisis estadístico comparativo

Presentamos un estudio empírico para comparar el comportamiento empírico de un grupo de algoritmos y responder a algunas cuestiones, que a partir de la experiencia práctica, nos hemos formulado:

(i) ¿Es posible determinar cuál es el mejor algoritmo en la práctica?

(ii) ¿Qué efectos producen los parámetros del problema (número de nodos, número de arcos, capacidad máxima, etc.) en el comportamiento empírico de los algoritmos?

(iii) ¿Qué parámetros, no presentes en la complejidad del caso peor, son significativos en el comportamiento empírico de un algoritmo?

(iv) Dado un algoritmo, ¿es su comportamiento independiente del generador de los problemas que se utiliza en el estudio?

(v) ¿Qué conjunto de algoritmos es el mejor en la práctica para una clase de problemas?

Para responder a estas cuestiones, hemos implementado, de una manera optimizada y contrastada, un grupo de algoritmos realizando un estudio práctico que desarrolla un diseño de experimento del tipo *split-plot*. Los datos generados han sido tratados mediante el procedimiento *anova* del módulo STAT del paquete estadístico SAS [66].

Nuestro estudio difiere de los estudios previos en que contiene un diseño de experimentos que tiene como variables respuesta el tiempo de CPU y otras como el recuento de las operaciones representativas. Mostraremos la dependencia de los algoritmos con respecto a los factores (capacidad máxima de los arcos, generador de redes, réplicas,...) e interacciones entre ellos que no han sido incluidos en otros estudios. Además, en nuestro estudio, el mayor ratio m/n es 50, mientras que en los previos era 20. Aunque las redes reales suelen ser poco densas, estamos interesados en mostrar el comportamiento empírico de los algoritmos para todos los casos. Todo esto nos permitirá clasificar a los algoritmos de acuerdo a los factores; más aún, podremos concluir si un algoritmo es sensible a un factor o a una combinación de ellos en la práctica.

En nuestro diseño experimental los objetivos se concretaron en diversas cuestiones. La primera fue estudiar los efectos de diversos factores sobre el tiempo de ejecución de un grupo numeroso de algoritmos. La segunda fue identificar la relativa eficiencia bajo las combinaciones de diferentes factores. Finalmente, la tercera fue estudiar los efectos de varios factores sobre el número de

operaciones necesitadas por los algoritmos para resolver el problema de flujo máximo.

De esta forma, obtenemos conclusiones acerca de grupos de algoritmos en relación a diferentes factores. Esto nos permite mostrar el comportamiento de las herramientas y las técnicas que los distintos algoritmos usan en la práctica.

2.1 Descripción básica de los algoritmos implementados

Previo al desarrollo del experimento realizado, es necesario dar una breve descripción de cada uno de los algoritmos incluidos en el mismo.

Algoritmo de Ford y Fulkerson [28]

Recordemos que este algoritmo es conocido como el algoritmo de caminos incrementales. El algoritmo procede identificando caminos incrementales y enviando flujo a través de dichos caminos hasta que la red residual no contiene tales caminos. Utiliza un recorrido en profundidad (DFS) para identificar un camino incremental en R en cada iteración.

La complejidad del algoritmo es $O(nmU)$. Denotaremos a este algoritmo por **F&F**.

Algoritmo de Dinic [25]

Dinic introduce el concepto de *red estratificada* L. La red estratificada $L=(V,A^*)$ es la red de caminos mínimos, donde A^* es el conjunto de arcos $(i,j) \in A$ que satisfacen la condición $d(i)=d(j)+1$, y todo camino del nodo fuente al nodo sumidero en L es un camino mínimo en R. El algoritmo de Dinic identifica varios caminos y envía flujo a través de todos ellos a la vez. Utiliza el concepto de *flujo bloqueante*: un flujo es un flujo bloqueante si no hay caminos incrementales en L. Un flujo máximo es bloqueante pero no al revés. La idea consiste en determinar un flujo bloqueante en la red estratificada y, entonces, actualizar el flujo parcial.

El algoritmo construye, a lo sumo, n redes estratificadas y encuentra un flujo bloqueante en un tiempo O(nm). Consecuentemente, la complejidad del algoritmo es O(n^2m). Denotamos este algoritmo por **DINIC**.

Algoritmo de Edmonds y Karp [26]

Es conocido como algoritmo de *caminos incrementales mínimos*. El método envía flujo a lo largo de un camino mínimo del nodo fuente al nodo sumidero en la red residual. La longitud del camino coincide con el número de arcos contenidos en él. El algoritmo utiliza un recorrido en amplitud (BFS) para identificar el camino mínimo. Edmonds y Karp muestran que para encontrar el flujo máximo en este caso se requieren, a lo sumo, $1/2mn$ envíos de flujo. Esto implica una complejidad O(nm^2). Denotamos a este algoritmo por **E&K**.

Algoritmo de Malhotra, Kumar y Maheshwari [56]

Es un algoritmo generalizado del de Dinic. Este método presenta una vía alternativa para obtener el flujo bloqueante. Introduce el concepto de *throughput* ("filtrado") de un nodo, que se define como la máxima cantidad de flujo que puede atravesar dicho nodo.

Formalmente: $thr(i) = \min\left\{ \sum_{j \in succ\{i\}} r_{ij}, \sum_{h \in pred\{i\}} r_{hi} \right\}$.

El algoritmo selecciona en cada iteración, el nodo i con el menor *throughput* ($thr(i)$). Entonces se sabe que es posible enviar $thr(i)$ unidades de flujo desde i a t y la misma cantidad desde s a i. El proceso se repite hasta que el *throughput* del nodo fuente o del nodo sumidero se hace igual a cero. En este caso el flujo bloqueante es determinado. La complejidad del algoritmo es O(n^3), debido a que el tiempo necesario para computar el flujo bloqueante es a lo sumo n^2. Denotaremos a este algoritmo por **MKM**.

Algoritmo de Goldberg [38]

Este método mantiene un preflujo como el definido por Karzanov [49]. La operación básica consiste en seleccionar un nodo activo y enviar flujo a sus vecinos. Para estimar los nodos activos

más cercanos al sumidero, el método usa las etiquetas distancias enviando flujo únicamente a través de arcos admisibles. Si el nodo activo seleccionado no tiene arcos admisibles, su etiqueta distancia es incrementada. A esta operación se la denomina *reetiquetar*. El algoritmo termina cuando la red no contiene nodos activos. La operación cuello de botella del algoritmo es el número de envíos no *saturantes*. Un envío es no saturante si $e(i) < r_{ij}$ para un nodo i (un envío es saturante si $e(i) \geq r_{ij}$). Por lo tanto, la complejidad resultante es $O(n^2m)$.

En la implementación de este procedimiento, computamos las etiquetas distancias exactas iniciales mediante un recorrido en amplitud (BFS). En este algoritmo es posible especificar diferentes reglas para seleccionar el nodo activo mejorando el número de envíos no saturantes, es decir, la complejidad. Los criterios son:

- Selección *Primero-en-entrar Primero-en-salir* (regla FIFO). Este criterio almacena los nodos activos en una cola y los selecciona de ella. La complejidad de este algoritmo es $O(n^3)$ (ver [38]). Denotamos a este algoritmo por **G_Q**.

- Selección *Último-en-entrar Primero-en-salir* (regla LIFO). Este criterio almacena los nodos activos en una pila y los selecciona de ella. Lo denotaremos por **G_S**.

- Combinación de las reglas FIFO y LIFO. Esta regla almacena los nodos activos en una cola doble; es decir, una cola con dos terminaciones. Los nodos son sacados del principio de la cola. Un nodo activo se añade al principio de la cola la primera vez que es activo; en otro caso, se añade al final de la cola. Denotamos al algoritmo por **G_DQ**.

- Selección de la Etiqueta Mayor. Esta regla consiste en seleccionar un nodo activo con la etiqueta distancia mayor. Para ello, se mantienen las listas $L(h) = \{i : i$ es activo y $d(i)=h\}$ gestionadas como colas. Cheriyan y Maheswari [19] muestran que esta regla implica una complejidad $O(n^2\sqrt{m})$. Denotamos al algoritmo por **G_HL**.

Algoritmo de Ahuja y Orlin [6]

Es una versión del algoritmo anterior. Usa una técnica de escalado del exceso para reducir los envíos no saturantes. La idea básica consiste en enviar flujo desde nodos activos con gran exceso a nodos con exceso pequeño. En este algoritmo, conocido como algoritmo de *escala en el exceso*, se define el *exceso dominante*, Δ, como el menor entero potencia de 2 que satisface $e(i) \leq \Delta,\ \forall i \in V - \{s,t\}$. Una iteración comienza con Δ igual al exceso dominante y finaliza cuando no hay nodos con un exceso mayor que $\Delta/2$. Entonces comienza la siguiente iteración con Δ igual a $\Delta/2$. Después de $\log U$ iteraciones, todos los nodos tienen un exceso igual a cero y, por tanto, se ha obtenido el flujo máximo. Para seleccionar un nodo activo con exceso mayor que $\Delta/2$ y con etiqueta distancia mínima, el algoritmo mantiene las listas: $L(r) = \{i \in V : e(i) > \Delta/2 \text{ y } d(i) = r\}$.

La complejidad de este algoritmo es $O(nm + n^2 \log U)$, donde $n^2 \log U$ es el número de envíos no saturantes para todas las fases y nm es una cota superior de las operaciones tales como envíos saturantes, operaciones de actualización de la etiqueta distancia y de determinar arcos admisibles. Este algoritmo es denotado por **A&O**.

Cod.	**Autor(es)**	**Complejidad**	**¿Etiqueta Distancia?**	**Ideas**
F&F	Ford-Fulkerson (1956)	$O(nmU)$	No	Caminos Incrementales (DFS)
E&K	Edmonds-Karp (1972)	$O(m^2 n)$	No	Caminos Incrementales Mínimos (BFS)
DINIC	Dinic (1970)	$O(mn^2)$	No	Red Estratificada
MKM	Malhotra-Kumar-Maheshwari (1978)	$O(n^3)$	No	Throughput
G_Q	Goldberg (1985)	$O(n^3)$	Sí	Preflujo; regla FIFO
G_DQ	Versiones del algoritmo de Goldberg	$O(mn^2)$	Sí	Preflujo; combinación regla LIFO y FIFO
G_S			Sí	Preflujo; regla LIFO
G_HL	Goldberg-Tarjan (1986), Cheriyan-Maheshwari (1989)	$O(n^2\sqrt{m})$	Sí	Preflujo; regla Mayor etiqueta distancia
A&O	Ahuja-Orlin (1989)	$O(mn + n^2 \log U)$	Sí	Escalado del exceso

Tabla 2.1. Algoritmos implementados.

Una clasificación de los anteriores algoritmos aparece en la Tabla 2.1. A continuación consideraremos las desventajas del uso de las etiquetas distancia y las estrategias propuestas para mejorar el comportamiento empírico del tiempo de ejecución de los algoritmos que las usan.

2.1.1 Estrategias para mejorar el comportamiento en la práctica de los algoritmos

Derigs y Meier [24], en un estudio computacional del algoritmo de Goldberg, muestran que los algoritmos realizan un gran número de pasos de actualización de las etiquetas distancias de los nodos para ser incrementadas a un valor mayor o igual a n. Este hecho ocurre porque es necesario identificar los nodos que no mejoran el valor del preflujo. El valor del preflujo máximo coincide con el preflujo de mayor cantidad de flujo enviado al sumidero. Este inmenso número de pasos de actualización de las etiquetas se realiza después de obtener el preflujo máximo y antes de obtener el flujo máximo.

Hay varias estrategias para mejorar el comportamiento práctico de los algoritmos de preflujo, es decir, disminuir el número de actualizaciones de la etiqueta distancia (ver Ahuja et al.[4]). Nosotros hemos usado las dos siguientes estrategias:

Computar_Etiquetas_Exactas

Este procedimiento consiste en realizar ocasionalmente un recorrido en amplitud inverso en la red residual, comenzando en el nodo sumidero. Si el nodo fuente no se alcanza, todos aquellos nodos que no tengan un camino hacia el sumidero incrementan su etiqueta distancia a un valor mayor o igual a n. Ejecutando esta operación cada m/b operaciones de *Envío/Reetiquetado*, para alguna constante b, no se incrementa la complejidad del peor caso de los algoritmos de preflujo. Esta estrategia es usada en los códigos desarrollados por Anderson y Setubal [7]. En nuestros códigos, el procedimiento *Computar_Etiquetas_Exactas* es llamado cada $m/2$ operaciones de *Envío/Reetiquetado*.

Estrategia de Derigs y Meier

Derigs y Meier [24], y Ahuja y Orlin [5], proponen la siguiente estrategia de manera separada. Sea x un preflujo y d una etiqueta

distancia válida con respecto a x. Entonces, llamamos a un número $z \in \{1,2,...,n-1\}$ un *gap* si las siguientes propiedades se satisfacen: $d(i) \neq z$ para todo i = 1,...,n y existe un nodo j con $d(j) > z$.

Si en un paso de reetiquetar se identifica un gap, entonces se realiza el siguiente paso de *reetiquetado-global*:

$$d(i) = \begin{cases} n & \text{si } d(i) > z \\ d(i) & \text{en otro caso} \end{cases} \quad \text{para todo } i = 1,\dots,n$$

Para implementar este procedimiento, mantenemos un vector de dimensión n, *Nnodos*, cuyos índices varían desde 0 a n-1. El valor de *Nnodos*(k) es el número de nodos cuya etiqueta distancia coincide con k. En un paso de reetiquetar de un nodo i, siempre se sustrae 1 de *Nnodos*(d(i)) y se examina si *Nnodos*(d(i))=0. Si *Nnodos*(d(i)) se hace igual a cero, entonces se realiza el reetiquetado-global, y el vector *Nnodos* es actualizado.

La estrategia de reetiquetado-global emplea un tiempo O(n). Por lo tanto, si se incluye en el procedimiento reetiquetar, la complejidad del algoritmo no aumenta.

Añadimos el sufijo 1 a los algoritmos que usan la estrategia *Computar_Etiquetas_Exactas* y el sufijo 2 a los algoritmos que usan la estrategia de Derigs y Meier. Hemos incorporado estas estrategias en los siguientes algoritmos:

		Computar_Etiquetas_Exactas	***Estrategia de Derigs y Meier***
Algoritmo	G_Q	G_Q1	G_Q2
	G_DQ	G_DQ1	G_DQ2
	G_S	G_S1	G_S2
	G_HL	G_HL1	G_HL2
	A&O	A&O1	A&O2

Tabla 2.2. Algoritmos con estrategias para mejorarlos

Todos los algoritmos han sido implementados en Pascal estándar usando las estructuras de datos requeridas para cada método en estudio y ejecutados en una estación de trabajo HP9000 715/33. Concretamente, los métodos estudiados son: F&F, E&K, DINIC, MKM, G_Q, G_Q1, G_Q2, G_DQ, G_DQ1, G_DQ2, G_HL, G_HL1, G_HL2, G_S, G_S1, G_S2, A&O, A&O1 y A&O2.

2.1.2 Generadores de Redes

Las complejidades teóricas de los algoritmos anteriores son funciones del número de nodos, número de arcos y la capacidad máxima de los arcos en la red. Por lo tanto, los casos particulares del problema de flujo máximo son redes generadas aleatoriamente con diferentes tamaños de esos parámetros. Hemos utilizado dos generadores: el generador NETGEN desarrollado por Klingman et al.. [52] y otro desarrollado por nosotros denominado generador FMGEN. No hemos usado el generador RMFGEN desarrollado por Goldfarb y Grigoriadis [40] debido a que no puede construir redes con un ratio $m/n > 6$.

En el generador NETGEN, el tamaño de la red se especifica mediante los siguientes parámetros: el número de nodos n, el número de arcos m, la capacidad mínima de un arco c_1 y la capacidad máxima de un arco c_2. El método genera una red aleatoria con las capacidades de los arcos uniformemente distribuidas en el intervalo $[c_1, c_2]$. En nuestro caso, el valor de c_1 es igual a 1 y el valor de c_2 coincide con U. Por lo tanto, el intervalo es $[1, U]$.

El generador FMGEN requiere los siguientes parámetros: número de nodos n, número de arcos m y la capacidad máxima de los arcos U. El generador FMGEN trabaja en dos fases. En la primera se construye un camino partiendo del nodo 1 que conecta todos los otros nodos (consideramos que siempre el nodo fuente es el 1 y el nodo sumidero es el n). Este proceso emplea n-1 arcos. Por lo tanto, en la segunda fase son generados aleatoriamente los $m-n+1$ arcos restantes. Las capacidades de los arcos están uniformemente distribuidas en el intervalo $[1, U]$.

El generador NETGEN y el generador FMGEN necesitan una semilla. Resumiendo, los parámetros necesarios para la especificación de ambos generadores son: *semilla*, n, m y U.

2.2 El Experimento

Esta sección introduce el experimento sobre el estudio de la resolución del problema de Flujo Máximo mediante los algoritmos

anteriormente mencionados. A continuación mostraremos las fases del experimento: *diseño, implementación* y *análisis*.

Seleccionamos como variables respuesta para comparar los algoritmos el tiempo de CPU para resolver cada problema. El tiempo de CPU no incluye el tiempo de los procedimientos de entrada/salida. Además, hemos considerado otras variables respuesta para los algoritmos que mantienen un preflujo:

- El número de envíos no saturantes (*NSAT*).
- El número de envíos saturantes (*SAT*).
- El número de operaciones de actualización de la etiqueta distancia (*RET*).

Los cinco factores estudiados en el experimento son *número de nodos* (n), *número de arcos* (m), *capacidad máxima* (U), *generador* (G) y el *algoritmo* (A) usado. La Tabla 2.3 muestra los niveles de n, m que dependen del número de nodos y el ratio m/n:

N	*m*	*m/n*
200	2000, 6000, 10000	10, 30, 50
500	5000, 15000, 25000	10, 30, 50
800	8000, 24000, 40000	10, 30, 50

Tabla 2.3. n, m y el ratio m/n.

El mayor valor para el ratio m/n es 50. Los niveles de la capacidad máxima (U) son 1, 10^4 y 10^8. Aquí queremos enfatizar que los estudios anteriores no utilizan este factor. El número de algoritmos que hemos incluido en el estudio es 18, debido a que hemos decidido eliminar el algoritmo F&F ya que necesita mucho más tiempo de CPU que los otros algoritmos. En resumen, tenemos cinco factores experimentales con niveles fijos.

2.3 El diseño

La mayor dificultad que tienen los investigadores en el uso del diseño *split-plot* es la de reconocer las restricciones sobre la aleatorización en el arranque experimental. En nuestro caso, hemos seleccionado aleatoriamente un nivel del número de nodos, un nivel de número de arcos y un nivel de capacidad máxima. A continuación

aplicamos aleatoriamente un generador y, seguidamente, resolvemos el problema especificado en orden aleatorio aplicando todos los algoritmos.

El orden experimental usado para obtener una especificación de un caso particular del problema de flujo máximo fue el siguiente: Una vez fijado el número de nodos, dependiendo de este valor, el número de arcos se obtuvo de manera aleatoria entre tres valores posibles. Por lo tanto, el número de arcos está anidado en el número de nodos. Finalmente, la capacidad máxima fue seleccionada aleatoriamente. Esta estructura básica fue replicada cinco veces.

Entonces, dados los parámetros de la red, se utilizan dos generadores: MFGEN y NETGEN. Para cada red previa, el correspondiente caso particular fue resuelto en orden aleatorio por todos los algoritmos, almacenando el tiempo de CPU y las otras variables respuestas para cada uno de ellos.

Las características del experimento conducen a un diseño *split-split-plot*. En nuestro caso, las combinaciones de número de nodos, número de arcos y capacidad máxima forman los tratamientos a aplicar a las 27 parcelas principales (main plots), 54 subparcelas (subplots) debidas al factor generador y las sub-subparcelas (sub-subplots) debidas al factor algoritmo (ver Milliken y Jonson [57]).

2.4 Análisis Estadístico I

Para responder a las cuestiones planteadas en la introducción, es necesario establecer comparaciones entre los niveles definidos por la situación experimental. Esto implica un análisis comprensivo de los efectos de los factores y sus interacciones sobre la variable respuesta. Por lo tanto, el objetivo fue identificar la importancia de los factores y sus interacciones en términos de la magnitud de sus efectos sobre los tiempos de CPU y las otras variables respuestas.

El modelo estadístico usado para reflejar el tiempo de CPU, con respecto a los factores y fuentes de error en este experimento y para el diseño split-split-plot, es el siguiente:

$$
\begin{aligned}
t_{ijkpql} &= \mu + n_i + m_{j(i)} + U_k + nU_{ik} + mU_{jk(i)} + R_q + \varepsilon_{ijkq} + G_p + nG_{ip} + mG_{jp(i)} + UG_{kp} \\
&+ nUG_{ikp} + mUG_{jkp(i)} + \varepsilon_{ijkpq} + A_l + nA_{il} + mA_{jl(i)} + UA_{kl} + nUA_{ikl} + mUA_{jkl(i)} + GA_{pl} \\
&+ nGA_{ipl} + mGA_{jpl(i)} + UGA_{kpl} + nUGA_{ikpl} + mUGA_{jkpl(i)} + \varepsilon_{ijkpql}
\end{aligned}
$$

donde

t_{ijkpql} es el tiempo de CPU.
μ es la media del tiempo de CPU.
n_i es el efecto del número de nodos, i=1,2,3.
$m_{j(i)}$ es el efecto del número de arcos dependiendo del i-ésimo nivel de nodos, j=1,2,3.
U_k es el efecto de la capacidad máxima, k=1,2,3.
G_p es el efecto del generador, p=1,2.
R_q es el efecto de la réplica, q=1,2,...,5.
A_l es el efecto del algoritmo, l=1,2,...,19.
ε_{ijkq} es el término del error plot.
ε_{ijkpq} es el término del error subplot.
ε_{ijkpl} es el término del error sub-subplot.

El método estadístico utilizado fue el análisis de la varianza. Este método da información para el contraste de las diferencias de las medias del tiempo de CPU bajo una única o múltiples combinaciones de tratamientos. Dado un diseño experimental, las diferencias significativas entre las medias de los tiempos de CPU de una combinación de tratamientos puede ser contrastada analizando las varianzas entre las medias.

Las hipótesis estadísticas pueden ser divididas en dos grupos principales: hipótesis para detectar diferencias significativas entre las medias para un único factor e hipótesis para detectar diferencias significativas entre las medias para interacciones de múltiples factores. Todas las hipótesis nulas pueden ser establecidas de la manera siguiente:

- Los valores de las medias poblacionales de los tiempos de CPU son iguales bajo los efectos de un único o varios factores.

- Las hipótesis alternativas se concretan en que en el conjunto de medias poblacionales, bajo las mismas combinaciones de tratamientos, hay al menos dos de ellas que no son iguales.

El nivel de significación fue previamente fijado en 5% para el análisis de la varianza. Hemos dividido los algoritmos en tres grupos, que son los siguientes:

Grupo-1: E&K, DINIC y MKM.

Grupo-2: G_Q, G_DQ, G_S, G_HL y A&O.

Grupo-3: G_Q1, G_Q2, G_DQ1, G_DQ2, G_HL1, G_HL2, G_S1, G_S2, A&O1 y A&O2.

Los algoritmos en el Grupo-1 no utilizan la función distancia. Los algoritmos en el Grupo-2 son los algoritmos originales de preflujo y los algoritmos en el Grupo-3 son los algoritmos anteriores donde se incluyen las estrategias para mejorarlos. Esta agrupación permite considerar los resultados de nuestro experimento en grupos y dar algunas conclusiones acerca de su comportamiento empírico en relación con las herramientas que utilizan.

La Tabla 2.4 resume la información dada por el procedimiento Anova para el análisis de los algoritmos del Grupo-2 y Grupo-3.

A partir de la Tabla 2.4 se puede observar que el comportamiento empírico de los algoritmos en el Grupo-2 y Grupo-3 únicamente dependen del número de nodos y el número de arcos. No dependen ni de la capacidad máxima, ni del generador ni de la réplica. En este punto es interesante remarcar que las complejidades teóricas de estos algoritmos únicamente dependen de n y m. Evidentemente, existe una dependencia del algoritmo que se utiliza.

2.5 Análisis Estadístico II: Comparación de medias

Cuando se compara más de dos medias, el procedimiento *Anova* muestra si estas son significativamente diferentes o no, pero no indica cuales de ellas son diferentes entre sí. Por ello es necesario realizar procedimientos de comparaciones múltiples entre las medias.

Fuente	Grados de libertad	Suma de Cuadrados	Media de cuadrados	Valor F	Pr > F
N	2	33962.72746	16981.36373	25.85	0.0001
m	6	20658.01716	3443.00286	5.24	0.0001
U	2	783.8016907	391.9008454	0.60	0.5525
nU	4	886.0985483	221.5246371	0.34	0.8523
mU	12	2073.905392	172.825449	0.26	0.9934
R	4	5938.181854	1484.545463	2.26	0.0676
G	1	184.4924136	184.4924136	0.33	0.5653
nG	2	1357.201428	678.600714	1.22	0.2982
mG	6	5698.796740	949.799457	1.71	0.1248
UG	2	160.3099543	80.1549771	0.14	0.8656
nUG	4	607.9195341	151.9798835	0.27	0.8942
mUG	12	684.0093725	57.0007810	0.10	0.9999
A	14	144335.4580	10309.6756	115.16	0.0001
nA	28	67168.7655	2398.8845	26.80	0.0001
mA	84	36855.8124	438.7597	4.90	0.0001
UA	28	1652.7149	59.0255	0.66	0.9133
nUA	56	1999.4700	35.7048	0.40	1.0000
mUA	168	4337.9240	25.8210	0.29	1.0000
GA	14	343.7669	24.5548	0.27	0.9963
nGA	28	3006.7153	107.3827	1.20	0.2163
mGA	84	11247.8709	133.9032	1.50	0.0026
UGA	28	377.6998	13.4893	0.15	1.0000
$nUGA$	56	1457.5020	26.0268	0.29	1.0000
$mUGA$	168	1915.5888	11.4023	0.13	1.0000
ε_{ijkq}	104	68314.9786	656.8748		
ε_{ijkpq}	108	59891.7506	554.5532		
ε_{ijkpql}	3024	270715.9310	89.5225		

Tabla 2.4. Análisis de la varianza de los tiempos de CPU de los algoritmos del Grupo-2 y Grupo-3.

Uno de los métodos más usados para comparar diversas medias es el método de *Diferencias Significativas de Tukey* (DST). Hemos aplicado este método para comparar y ordenar los efectos de los niveles de un único factor en el tiempo CPU bajo combinaciones de tratamientos. Este procedimiento controla la tasa de error experimental (TEE) definida como la probabilidad de rechazar una o más hipótesis nulas cuando aplicamos tests estadísticos con dos o más hipótesis nulas.

El método DST fue principalmente aplicado a los factores y las interacciones de mayor interés. Los resultados de este método, para el factor algoritmo, se muestran en la Tabla 2.5. Se ha de entender en dicha tabla que *algoritmos asociados con una misma letra no son significativamente diferentes*. Las letras A, B, y C indican una clasificación (grupos de Tukey) en orden no creciente del

mencionado factor con respecto a la media del tiempo de CPU en segundos.

Grupo	*Media CPU*	*Algoritmo*	*Grupo*	*Media CPU*	*Algoritmo*
A	17,005	A&0	C	0,352	G_HL1
B	11,817	G_S	C	0,095	A&O1
B	11,578	G_HL	C	0,082	A&O2
B	11,222	G_Q	C	0,077	G_Q2
B	11,200	G_DQ	C	0,076	G_S2
C	0,417	G_DQ1	C	0,076	G_DQ2
C	0,414	G_Q1	C	0,065	G_HL2
C	0,368	G_S1			

Tabla 2.5. Test de Tukey para el análisis de la varianza del tiempo de CPU en relación al factor algoritmo para el Grupo-2 y el Grupo-3.

Podemos deducir a partir de la Tabla 2.5 que, en nuestra experimentación, los algoritmos que no utilizan estrategias para mejorar el número de actualizaciones de la etiqueta distancia emplean mucho más tiempo que los algoritmos del Grupo-3, que si las incorporan. Debido a esto, hemos decidido no incluir los algoritmos del Grupo-2 en los análisis siguientes. En la Tabla 2.5, se puede observar que no hay diferencias significativas entre los algoritmos del Grupo-3.

La Tabla 2.6 resume la información proporcionada por el procedimiento *Anova* para el análisis de los algoritmos en el Grupo-1 y en el Grupo-3. Se puede observar, a partir de esta tabla, que el comportamiento empírico de los algoritmos en el Grupo-1 y en el Grupo-3 depende únicamente del número de nodos, número de arcos, la capacidad máxima y el generador. No depende de la réplica. Otra vez, lógicamente, depende del algoritmo usado.

Los resultados del método DST para el factor *algoritmo* se muestran en la Tabla 2.7. Las letras A, B, C, D y E indican una clasificación en orden no creciente del mencionado factor con respecto a la media del tiempo de CPU en segundos. Podemos deducir de la Tabla 2.7 que el algoritmo E&K requiere el mayor tiempo de CPU en comparación con los otros. Además, no hay diferencias significativas entre los siguientes algoritmos: MKM, GDQ1, G_Q1, G_S1 y G_HL1. A partir de está tabla se puede observar que no hay diferencias significativas entre los algoritmos que utilizan la estrategia de Derigs y Meier y los algoritmos A&O1 y DINIC. En nuestro experimento computacional, el menor tiempo medio de CPU lo alcanza el algoritmo GHL2.

Fuente	Grados de libertad	Suma de Cuadrados	Media de cuadrados	Valor F	Pr > F
n	2	90.41334324	45.20667162	32.18	0.0001
m	6	137.3028228	22.8838038	16.29	0.0001
U	2	24.04687983	12.02343991	8.56	0.0004
nU	4	7.60525551	1.90131388	1.35	0.2553
mU	12	13.49218811	1.12434901	0.80	0.6492
R	4	4.46105281	1.11526320	0.79	0.5317
G	1	4.23010433	4.23010433	3.73	0.0561
nG	2	1.55965403	0.77982702	0.69	0.5050
mG	6	3.75189488	0.62531581	0.55	0.7681
UG	2	5.42335250	2.71167625	2.39	0.0964
nUG	4	3.26523461	0.81630865	0.72	0.5803
mUG	12	10.51640530	0.87636711	0.77	0.6772
A	12	538.7731806	44.8977651	53.65	0.0001
nA	24	136.1374579	5.6723941	6.78	0.0001
mA	72	247.4702236	3.4370864	4.11	0.0001
UA	24	165.9201851	6.9133410	8.26	0.0001
nUA	48	73.8800266	1.5391672	1.84	0.0004
mUA	144	156.8396541	1.0891643	1.30	0.0107
GA	12	72.9115261	6.0759605	7.26	0.0001
nGA	24	29.1351525	1.2139647	1.45	0.0725
mGA	72	52.7132677	0.7321287	0.87	0.7645
UGA	24	42.1498383	1.7562433	2.10	0.0014
$nUGA$	48	47.0245672	0.9796785	1.17	0.1977
$mUGA$	144	102.2873972	0.7103291	0.85	0.9005
ε_{ijkq}	104	146.0930205	1.4047406		
ε_{ijkpq}	108	122.5128860	1.1343786		
ε_{ijkpql}	2592	2169.109279	0.836848		

Tabla 2. 6. Análisis de la varianza de los tiempos de CPU de los algoritmos del Grupo-1 y del Grupo-3.

Grupo					Media	Algoritmo
				A	1,5669	E&K
			B		0,5815	MKM
		C	B		0,4166	G_DQ1
		C	B		0,4139	G_Q1
		C	B		0,3684	G_S1
	D	C	B		0,3522	G_HL1
E	D	C			0,2474	DINIC
E	D				0,0947	A&O1
E					0,0820	A&O2
E					0,0765	G_Q2
E					0,0763	G_DQ2
E					0,0756	G_S2
E					0,0653	G_HL2

Tabla 2.7. Test de Tukey para el análisis de la varianza del tiempo de CPU en relación al factor algoritmo para el Grupo-1 y elGrupo-3.

En la Tabla 2.8 se presenta una clasificación con respecto a la combinación de los factores número de nodos y algoritmo. Como siempre, las letras A, B, C, etc., indican una clasificación en orden no creciente con respecto a la media del tiempo de CPU. De esta

tabla se deduce que no hay diferencias significativas entre los algoritmos del Grupo-3 y los algoritmos DINIC y MKM, para un número de nodos igual a 200. Se puede observar que, para todos los valores de nodos, no existe diferencia significativa entre los siguientes algoritmos: DINIC, G_HL1, A&O1, G_Q2, G_DQ2, G_HL2, G_S2 y A&O2. Esto implica que el factor número de nodos tiene una influencia similar sobre dichos algoritmos. Finalmente, el algoritmo E&K es significativamente diferente al resto de algoritmos para cualquier valor del numero de nodos.

En la Tabla 2.9, se muestra una clasificación para la combinación de los factores algoritmo y capacidad máxima. Todos los algoritmos son iguales para la capacidad máxima igual a 1. Cuando los valores de la capacidad son mayores que 1, aparecen diferencias significativas entre los algoritmos, pero estas diferencias son las mismas para todos los valores de la capacidad máxima. Podemos concluir que el factor capacidad máxima mantiene la clasificación de los algoritmos. El algoritmo E&K, otra vez, es muy diferente al resto de los algoritmos. Deducimos de la Tabla 2.9 que el peor algoritmo para capacidades mayores que 1 es E&K.

	Nodos								
Algoritmo	200			500			800		
E&K		A				A			A
DINIC	B	A	D	C	B		C		
MKM	B	A			B			B	
G_Q1	B	A			B		C	B	
G_DQ1	B	A			B		C	B	
G_HL1	B	A	D	C	B		C		
G_S1	B	A		C	B		C		
A&O1	B	A	D	C			C		
G_Q2	B		D	C			C		
G_DQ2	B		D	C			C		
G_HL2	B		D				C		
G_S2	B		D	C			C		
A&O2	B		D	C			C		

Tabla 2.8. Test de Tukey para el análisis de la varianza del tiempo de CPU con respecto a los factores algoritmo y número de nodos.

	Capacidad Máxima						
Algoritmo	1	10e+4			10e+8		
E&K	A			A			A
DINIC	A	C	B		C	B	
MKM	A		B			B	
G_Q1	A	C	B		C	B	
G_DQ1	A	C	B		C	B	
G_HL1	A	C	B		C	B	
G_S1	A	C	B		C	B	
A&O1	A	C			C		
G_Q2	A	C			C		
G_DQ2	A	C			C		
G_HL2	A	C			C		
G_S2	A	C			C		
A&O2	A	C			C		

Tabla 2.9. Test de Tukey para el análisis de la varianza del tiempo de CPU con respecto a los factores algoritmo y capacidad máxima.

La Tabla 2.10 muestra una clasificación para la combinación de los factores algoritmo y generador. Se puede observar que la clasificación de los algoritmos cambia cuando se consideran

diferentes generadores. Para el generador NETGEN, no existe una diferencia significativa entre los algoritmos en el Grupo-3 y DINIC. Sin embargo, cuando se considera el generador MFGEN, la clasificación de los algoritmos es más compleja. En este caso, el comportamiento empírico de los algoritmos MKM, G_Q1 y G_DQ1 es el mismo e igualmente ocurre con los algoritmos DINIC, G_HL1, A&O1, G_Q2, G_DQ2, G_HL2, G_S2 y A&O2. Con independencia del generador utilizado, el algoritmo E&K es el peor y el que presenta más diferencias respecto a los demás.

Creemos necesario comentar que, debido al hecho de que el comportamiento de los algoritmos es sensible al generador, los grafos deberían ser explicados mediante más parámetros que el número de nodos, número de arcos, y capacidad máxima.

	Generador						
Algoritmo	*FMGEN*				*NETGEN*		
E&K				A			A
DINIC	D	C	B		C		
MKM			B			B	
G_Q1			B		C	B	
G_DQ1			B		C	B	
G_HL1	D	C	B		C		
G_S1		C	B		C		
A&O1	D	C			C		
G_Q2	D	C			C		
G_DQ2	D	C			C		
G_HL2	D				C		
G_S2	D	C			C		
A&O2	D	C			C		

Tabla 2.10. Test de Tukey para el análisis de la varianza del tiempo de CPU con respecto a los factores algoritmo y generador.

2.6 Análisis estadístico de los algoritmos individualmente

En las tablas anteriores, hemos obtenido clasificaciones de los algoritmos con respecto a los factores o combinaciones de factores con influencia significativa. Aunque, con menor efecto comparativo, pensamos que podría ser apropiado estudiar individualmente cada uno de los algoritmos que pertenecen al Grupo-1 y al Grupo-3, con respecto a los siguientes factores: nodos, arcos, capacidad máxima y generador. En este caso, el modelo estadístico del *split-plot* es el siguiente:

$$t_{ijkpql} = \mu + n_i + m_{j(i)} + U_k + nU_{ik} + mU_{jk(i)} + R_q + \varepsilon_{ijkq} + G_p + nG_{ip} + mG_{jp(i)}$$
$$+ UG_{kp} + nUG_{ikp} + mUG_{jkp(i)} + \varepsilon_{ijkpq}$$

donde:

t_{ijkpq} es el tiempo de CPU.
μ es la media del tiempo de CPU.
n_i es el efecto del número de nodos, $i=1,2,3$.
$m_{j(i)}$ es el efecto del número de arcos para el i-ésimo nivel de nodos, $j=1,2,3$.
U_k es el efecto de la capacidad máxima, $k=1,2,3$.
G_p es el efecto del generador, $p=1,2$.
R_q es el efecto de la réplica, $q=1,2,...,5$.
ε_{ijkp} es el término de error plot.
ε_{ijkpq} es el término de error subplot.

Un resumen de la información proporcionada por el procedimiento *Anova* para el análisis individual de los algoritmos en el Grupo-1 y en el Grupo-3 aparece en la Tabla 2.11. La influencia de los factores o combinaciones de los factores para cada algoritmo aparece en esta tabla. La presencia del símbolo ✓ significa que el factor influye en el algoritmo.

	n	*m*	*U*	*R*	*G*	*nU*	*nG*	*mG*	*UG*
E&K	✓	✓	✓		✓				
DINIC	✓	✓	✓		✓	✓	✓	✓	✓
MKM	✓	✓	✓		✓	✓	✓	✓	✓
G_Q1	✓	✓							
G_Q2	✓	✓	✓					✓	
G_DQ1	✓	✓							
G_DQ2	✓	✓	✓					✓	
G_HL1	✓	✓			✓				
G_HL2	✓	✓	✓		✓	✓	✓	✓	✓
G_S1	✓	✓							
G_S2	✓	✓	✓		✓			✓	
A&O1	✓	✓	✓		✓	✓	✓	✓	✓
A&O2	✓	✓	✓		✓	✓	✓	✓	✓

Tabla 2.11. Análisis de la varianza del tiempo de CPU para cada algoritmo.

Se deduce de la Tabla 2.11, que todos los algoritmos dependen del número de nodos, número de arcos, pero no de la réplica. Los algoritmos G_Q1, G_DQ1, G_HL1 y G_S1 no dependen de la capacidad máxima. Sin embargo, los algoritmos DINIC, MKM, G_HL2, A&O1 y A&O2 dependen de todos los factores (con la excepción de la réplica). En este caso, es interesante comentar que el algoritmo G_HL2 es el mejor en la práctica aún cuando es

sensible a todos los factores. Los algoritmos G_Q1, G_Q2, G_DQ1, G_DQ2 y G_S1 no dependen del generador. Este hecho implica que estos algoritmos son más robustos en la práctica. En particular, esto es interesante para los algoritmos G_Q2 y G_DQ2, dado que son buenos algoritmos en la práctica (ver Tabla 2.5 y Tabla 2.7). Los algoritmos G_Q1, G_DQ1 y G_S1 únicamente dependen del número de nodos y del número de arcos. Una posible explicación es que la influencia de estos factores es mucho mayor que la de los otros.

Finalmente, es difícil comparar la complejidad teórica de los algoritmos con el comportamiento empírico de los mismos. Por ejemplo, la complejidad del caso peor de todos los algoritmos estudiados no depende de la capacidad máxima; sin embargo, esto no es verdad en la experiencia práctica. Además, el comportamiento empírico de los algoritmos depende del generador usado. Sin embargo, la complejidad teórica no contiene parámetros que tengan en cuenta, por ejemplo, la morfología de la red.

2.7 Análisis estadístico para las variables respuesta NSAT, SAT y RET

En nuestro experimento, hemos decidido contar el número de operaciones cuello de botella realizadas por los algoritmos del Grupo-3. Las operaciones examinadas son el número de envíos no saturantes ($NSAT$), el número de envíos saturantes (SAT) y el número de actualizaciones de la etiqueta distancia (RET).

Para estas variables respuestas hemos utilizado el modelo estadístico introducido en la sección *análisis estadístico I*. Hemos aplicado el logaritmo a estas variables siguiendo la familia de transformaciones de *Box* y *Cox* [12], para romper la relación lineal entre las medias y la desviación estándar en cada parcela (plot).

Las dependencias de las variables $NSAT$, SAT y RET, con respecto a los factores del diseño y sus combinaciones, aparecen en la Tabla 2.12. Únicamente se listan las dependencias significativas.

	NSAT	*SAT*	*RET*
n	✓	✓	✓
m	✓	✓	✓
U	✓		✓
Un	✓		
G	✓	✓	✓
nG	✓	✓	✓
mG	✓	✓	✓
UG	✓		

	NSAT	*SAT*	*RET*
A	✓	✓	✓
nA	✓	✓	✓
mA	✓	✓	✓
UA	✓	✓	✓
GA		✓	✓
nGA	✓	✓	✓
mGA	✓	✓	✓
UGA	✓		

Tabla 2.12. Análisis de la varianza para NSAT, SAT y RET.

A partir de la Tabla 2.12 se puede deducir que las variables SAT y RET tienen las mismas dependencias. Esto es claramente debido al hecho de que los algoritmos realizan la actualización de una etiqueta distancia después de que se realiza un envío saturante. La variable *NSAT* es más sensible que *SAT* y *RET* al factor capacidad máxima y combinaciones con este. Estas variables no dependen del factor réplica; dependen principalmente del número de nodos, número de arcos, del algoritmo y del generador, como ocurre con la variable tiempo de CPU.

Finalmente, mostramos el orden resultante del test de Tukey para cada variable con respecto al factor algoritmo. La clasificación para *NSAT*, *SAT* y *RET* para cada algoritmo se muestra en la Tabla 2.13.

En este punto queremos remarcar dos aspectos. Primero, la aplicación del logaritmo a NSAT, SAT y RET. Segundo, cuando la capacidad máxima es igual a 1, todos los envíos son saturantes; por lo tanto, la media del valor de SAT es mayor que la media del valor de NSAT.

En la Tabla 2.13, se puede observar que los algoritmos G_Q1, G_DQ1, G_S1 y G_HL1 son significativamente diferentes a los otros. Además, estos realizan un mayor número de envíos saturantes, envíos no saturantes y actualizaciones de las etiquetas que el resto de algoritmos. Los valores de *NSAT*, *SAT* y *RET* son equivalentes para los algoritmos G_Q2, G_DQ2 y G_S2. La mejor de las versiones del algoritmo de Goldberg es G_HL2. Los algoritmos A&O1 y A&O2 realizan el menor número de estas operaciones. Esto es importante si recordamos que el algoritmo A&O se diseñó para disminuir el número de envíos no saturantes con respecto al algoritmo de Goldberg. En particular, A&O1 realiza el menor

número de estas operaciones. Sin embargo, este hecho no se refleja en el tiempo de CPU. Los algoritmos A&O1 y A&O2 son rápidos, pero no son los mejores. Esto es debido a que el algoritmo de Ahuja y Orlin emplea mucho más tiempo que el algoritmo de Goldberg en la operación de seleccionar arcos admisibles.

	NSAT		*SAT*		*RET*	
Algoritmo	*Grupo*	*Media*	*Grupo*	*Media*	*Grupo*	*Media*
G_Q1	A	5,0765	A	6,2226	A	6,1111
G_DQ1	A	5,0942	A	6,2320	A	6,1043
G_S1	A	5,1549	A	6,1263	A	6,0404
G_HL1	A	5,0863	A	6,1847	A	6,0178
A&O1	D	4,0205	D	4,7616	D	4,2571
G_Q2	C B	4,3887	C B	5,3748	B	5,2067
G_DQ2	C B	4,4256	B	5,3961	B	5,2115
G_S2	B	4,5226	C B	5,2547	B	5,2106
G_HL2	C	4,3020	C	5,2310	C	4,8907
A&O2	C	4,3138	D	4,8436	C	4,7888

Tabla 2.13. Test de Tukey para el análisis de la varianza de NSAT, SAT y RET con respecto al factor algoritmo.

2.8 Resumen de conclusiones

Los resultados de los experimentos anteriores nos permiten obtener algunas conclusiones acerca del comportamiento empírico de los algoritmos. El análisis determina, de una manera clara, qué algoritmos son en la práctica los más eficientes, en un sentido global y con relación a los parámetros del problema. Resumimos nuestras conclusiones en los siguientes puntos:

- La complejidad de los algoritmos basada en el caso peor es insuficiente para clasificar el comportamiento de los algoritmos en la práctica. Además, algunos factores que influyen en el algoritmo no están incluidos en la función de la complejidad teórica.

- En nuestro experimento las estrategias para mejorar el número de actualizaciones de la etiqueta distancia emergen como necesarias. En particular, la estrategia de Derigs y Meier es mejor que la estrategia *Computar_Etiquetas_Exactas* (Tabla 2.5). Además, el número de envíos saturantes, envíos no saturantes y actualizaciones de la etiqueta distancia son mayores para la estrategia *Computar_Etiquetas_Exactas* que para la estrategia de Derigs y Meier (Tabla 2.13).

•El mejor de los algoritmos de caminos incrementales es DINIC. El mejor de los algoritmos de preflujo es G_HL2. Además, el algoritmo G_HL2 alcanza el menor tiempo medio de CPU, pero no hay diferencias significativas entre los algoritmos G_HL2, G_S2, G_DQ2, G_Q2, A&O2 y A&O1 (Tabla 2.7).

•Los factores nodos y arcos influyen en todos los algoritmos. En particular, los algoritmos G_Q1, G_DQ1 y G_S1 únicamente dependen de estos factores (Tabla 2.11). Sin embargo, el factor nodos tiene una influencia similar sobre los algoritmos DINIC, G_HL1, A&O1, G_Q2, G_DQ2, G_HL2, G_S2 y A&O2, debido a que estos no son diferentes para todos los valores del número de nodos (Tabla 2.8).

•La máxima capacidad influye en el comportamiento en la práctica de los algoritmos. Con la excepción del algoritmo A&O1, todos los algoritmos que usan la estrategia *Computar_Etiquetas_Exactas* no dependen de la capacidad máxima (Tabla 2.11). Sin embargo, el agrupamiento de Tukey es el mismo para valores de capacidad mayores que 1. Esto significa que para cada algoritmo, en el correspondiente grupo, este factor tiene un efecto similar (Tabla 2.9).

•Los generadores de redes que se utilizan en el experimento tienen una influencia sobre el comportamiento empírico de los algoritmos. Además, el agrupamiento de Tukey cambia cuando se considera un generador particular (Tabla 2.10). Debido a esto, en el estudio experimental parece necesario incluir algunos parámetros relacionados con la morfología de la red para explicar el comportamiento práctico de un algoritmo. Sin embargo, si un buen algoritmo no depende del generador en la práctica, podemos decir que este algoritmo es robusto. En nuestro experimento, G_Q1, G_DQ1, G_S1, G_Q2 y G_DQ2 son robustos (Tabla 2.11). En particular, los algoritmos G_Q2 y G_DQ2 son rápidos y robustos.

•El número de envíos saturantes y actualizaciones de la etiqueta distancia dependen casi de los mismos factores. Estas operaciones son menos sensibles a la capacidad máxima que el número de envíos no saturantes (Tabla 2.12). El número de

envíos saturantes, envíos no saturantes y actualización de las etiquetas distancia son equivalentes para los algoritmos que utilizan la estrategia *Computar_Etiquetas_Exactas*, con la excepción del algoritmo A&O1. Cuando se considera la estrategia de Derigs y Meier, la clasificación resultante es más compleja. Los algoritmos A&O1 y A&O2 son los que realizan un menor número de envíos no saturantes, saturantes y actualizaciones de la etiqueta distancia (Tabla 2.13). Este hecho concuerda con la intención en el diseño original del método.

Nuestra intención, a la hora de realizar un estudio experimental sobre un conjunto extenso de algoritmos para el problema de flujo máximo, además de responder a las cuestiones previamente planteadas, se dirigía a adquirir un conocimiento preciso de los aspectos computacionales de los algoritmos disponibles. ¿Con qué fin? Con el fin de obtener, si fuera posible, un algoritmo mejor que los existentes. El problema de flujo máximo desde el punto de vista computacional está abierto, debido a que los investigadores no desechan la posibilidad de determinar un algoritmo que resuelva el problema en un tiempo O(nm). Hasta hoy, no existe un algoritmo que alcance esta complejidad para cualquiera que sea el tamaño de la red.

En la siguiente sección presentamos un nuevo algoritmo para resolver el problema de flujo máximo en un tiempo de O($nm\log(\frac{U}{n})$). Nuestro principal resultado es que bajo la asunción de que $U = O(n)$, nuestro algoritmo tiene complejidad O(nm) para cualquier valor de n y m. Ningún otro algoritmo de la literatura consultada alcanza esta complejidad para cualesquiera que sean n y m bajo la misma hipótesis. Si $U = O(n^K)$ para algún K >1, el algoritmo corre en un tiempo O($nm\log n$) coincidiendo con el algoritmo de Sleator y Tarjan [73], sin utilizar estructuras de datos complejas. Además, para redes con capacidades unitarias el algoritmo corre en O($n^{2/3}m$). Además, dicho algoritmo posibilita la inclusión de las estrategias ya comentadas, así como otras propias, que redundan en una mejora práctica del comportamiento empírico.

3. Algoritmo de dos fases escalado en las capacidades

En este apartado proponemos un algoritmo de escalado en las capacidades que utiliza el método de caminos incrementales mínimos de Ahuja y Orlin [5] en una primera fase (dado en el capítulo 1) y el algoritmo de Ford y Fulkerson [28] en la segunda fase, de tal manera que se obtiene un algoritmo de complejidad $O(nm \log(\frac{U}{n}))$.

3.1 Explicación del algoritmo de dos fases escalado en las capacidades

El primer algoritmo de escalado fue propuesto por Gabow [31] y requiere un tiempo $O(nmlogU)$. Ahuja y Orlin [5] también proponen un algoritmo escalado de la misma complejidad.

Para definir el escalado en las capacidades introducimos un parámetro Δ y nos referimos a la red *Δ-residual*, $R(\Delta)$, como a la red que únicamente contiene arcos cuya capacidad residual es al menos Δ. De esta manera, todos los caminos incrementales en $R(\Delta)$ tienen una capacidad residual de al menos Δ. Se dice que un flujo es *Δ-óptimo* si en R no existe ningún camino incremental con capacidad residual de al menos Δ.

El algoritmo trabaja en fases de escalado. Cada fase tiene un Δ fijo y, cuando finaliza una fase, comienza la siguiente con $\Delta=\Delta/2$. Al principio $\Delta = 2^{\lfloor \log U \rfloor + 1}$, es decir, es igual al menor entero potencia de dos que es mayor o igual que U. La última fase del algoritmo ocurre cuando $\Delta=1$. En esta última fase $R(\Delta)=R$ y, por tanto, al finalizar el algoritmo se determina un flujo máximo. De esta manera, el método realiza $\lfloor \log U \rfloor + 1$ fases de escalado.

En cada fase Δ, el método trabaja a su vez en dos fases. En la primera fase, se aplica el algoritmo de caminos mínimos incrementales (capítulo 1), parando cuando la etiqueta distancia del nodo fuente es mayor o igual que $K(\Delta)$ (el valor de $K(\Delta)$ será fijado posteriormente). En esta fase obligamos a que cada envío de flujo lleve, exactamente, Δ unidades de flujo; por tanto, en el algoritmo anterior no hace falta el procedimiento *Aumentar*,

consiguiendo que el esfuerzo realizado en cada envío sea de O(1). Esto es así ya que cuando realizamos un procedimiento *Avanzar*(i), a través del arco admisible actual (i, j), actualizamos las capacidades residuales haciendo $r_{ij} = r_{ij} - \Delta$ y $r_{ji} = r_{ji} + \Delta$; y cuando realizamos un paso *Retroceder*(i) actualizamos las capacidades residuales haciendo $r_{pred(i)i} = r_{pred(i)i} + \Delta$ y $r_{i\,pred(i)} = r_{i\,pred(i)} - \Delta$. Por tanto, si (i, j) está en un camino admisible, ya hemos enviado las Δ unidades de flujo y, si retrocedemos sobre él, deshacemos ese envío.

En la segunda fase utilizamos el algoritmo de Ford y Fulkerson, es decir, se identifican caminos incrementales desde la fuente al sumidero en $R(\Delta)$, mediante un recorrido en profundidad, y se envía por cada uno de ellos una cantidad de flujo igual a la capacidad residual de este, que, como ya hemos mencionado, debe ser mayor o igual a Δ (y menor estricto que 2Δ). Al final hemos obtenido un flujo Δ-óptimo.

El método que proponemos exige, además, modificaciones en el algoritmo de caminos incrementales mínimos. Para llevarlas a cabo notemos que un arco (i, j) es *Δ-admisible* si $d(i) = d(j) + 1$ y $r_{ij} \geq \Delta$. El método de caminos incrementales mínimos procederá, por lo tanto, identificando caminos incrementales *Δ-admisibles*. También debe modificarse el procedimiento *Retroceder*(i), de tal manera que la etiqueta distancia, una vez actualizada, sea $d(i) := \min\{\, d(j) + 1 : (i, j) \in A(i) \text{ y } r_{ij} \geq \Delta\}$. Según ya hemos mencionado con anterioridad, como obligamos a enviar Δ unidades de flujo por cada camino incremental, no se utilizará el procedimiento *Aumentar*. En cambio, debemos utilizar una función *Enviar_Flujo* que es la encargada de actualizar las capacidades residuales de los arcos del camino incremental parcial actual. Por último, el criterio de parada es ahora $d(s) \geq K(\Delta)$.

Después de todos estos comentarios, el algoritmo de caminos incrementales mínimos de capacidad residual Δ es el siguiente:

```
Procedure Enviar_Flujo(i,j,Δ);
  begin
  r_ij = r_ij - Δ ; r_ji = r_ji + Δ
  end;
```

```
Algoritmo Caminos_Incrementales_Mínimos_De_Capacidad(Δ,K(Δ));
  begin
  Sea X el vector de flujos actual; d(t):= 0;
  Obtener las etiquetas distancias exactas mediante un recorrido
  en amplitud inverso en la red Δ-residual empezando en el nodo
  sumidero t;
  i := s;
  while d(s) < K(Δ) do
    if i tiene un arco Δ-admisible then
      begin
      Avanzar(i);
      Enviar_Flujo(pred(i),i, Δ);
      if i = t then i = s
      end
    else
      begin
      if i ≠ s then Enviar_Flujo(pred(i),i,- Δ);
      Retroceder(i);
      end
  end.
```

En la segunda fase únicamente hay que modificar el recorrido en profundidad para que sólo tenga en cuenta a los arcos que tienen una capacidad mayor o igual que Δ. Por todo ello, el algoritmo de dos fases escalado queda de la siguiente manera:

```
Algoritmo De_Dos_Fases_Escalado(2FEC);
  begin
  X:=0;
  Δ := 2^⌈log U⌉;
  while Δ ≥ 1 do
    begin
    Caminos_Incrementales_Mínimos_de_Capacidad(Δ,K(Δ));{1ª fase}
    while Exista una camino P de s a t en R(Δ) do {2ª fase}
      begin
      δ := min{r_ij : (i, j) ∈ P};
      Envía δ unidades de flujo a lo largo de P
      end;
    Δ := Δ / 2
    end
  end
```

3.2 Un ejemplo

La Figura 2.1 ilustra los pasos del algoritmo (suponemos que fijamos $K(\Delta) = 3$ para todas las fases de escalado). La red de

partida tiene un valor de U = 4 y, por tanto, al principio Δ = 4. Como se puede constatar, no hay caminos incrementales cuya capacidad residual sea mayor o igual que 4. La siguiente fase de escalado comienza con Δ = 2. En esta fase de escalado el único camino incremental tiene longitud 2 y, como $K(2)$ = 3, el algoritmo de caminos incrementales mínimos identificará el camino *s*-*a*-*t* y enviará 2 unidades de flujo a través de él. En este momento no hay caminos incrementales de capacidad residual mayor o igual que 2. A continuación comenzaría la última fase de escalado con Δ = 1. Se puede observar que en este caso $R(1)=R$ y que existen 3 posibles caminos incrementales: dos de ellos de longitud 2 y uno de longitud 3. Como $K(1)$ = 3, el algoritmo identificaría primero los caminos *s*-*a*-*t* y *s*-*b*-*t* y enviaría 1 unidad de flujo por cada uno de ellos (ambos caminos son arco-disjuntos). En este momento no hay más caminos incrementales y, por lo tanto, el flujo es máximo. Hemos de añadir, que si el camino *s*-*b*-*a*-*t* tuviese que haber sido considerado en alguna fase de escalado del algoritmo, como tiene una longitud igual a 3, se hubiese identificado en la segunda fase del método (recorrido en profundidad).

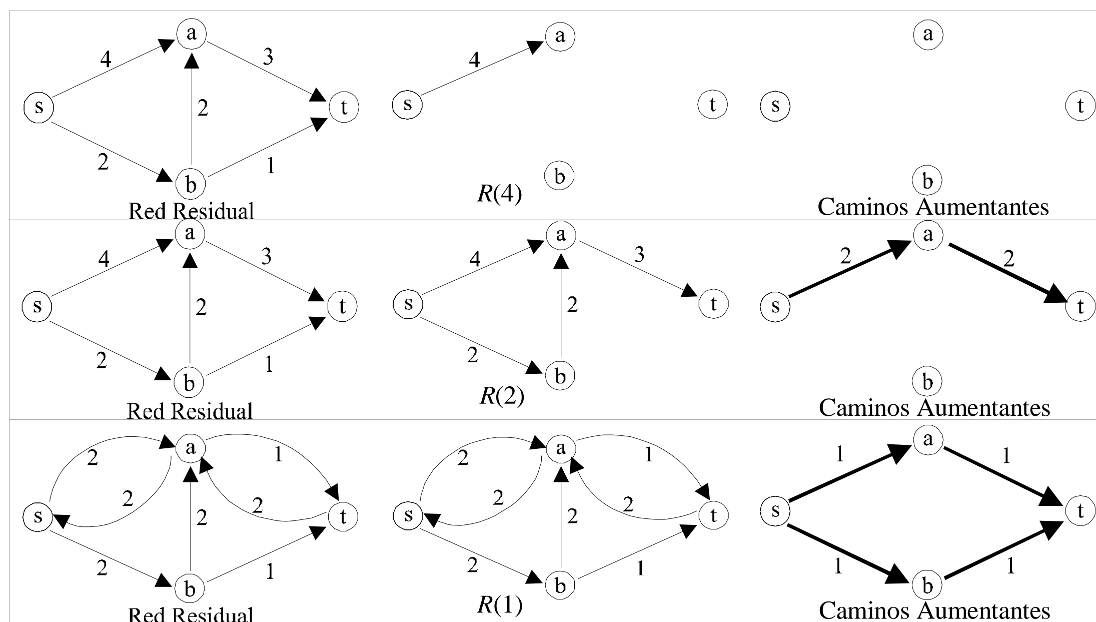


Figura 2.1. Red ejemplo.

En el cálculo de la complejidad involucraremos a $K(\Delta)$, y obtendremos el valor para el que se hace mínima. Dada una fase de escalado Δ, el algoritmo trabaja en dos fases como ya hemos mencionado. En la primera fase se ejecuta el método de caminos incrementales mínimos de capacidad Δ hasta que la etiqueta distancia del nodo fuente s sea mayor o igual que $K(\Delta)$. En la segunda fase se identifican caminos incrementales hasta obtener un flujo *Δ-óptimo*. Las correspondientes complejidades en cada fase se obtendrán en función de $K(\Delta)$.

Lema 2.1. *La primera fase reliza, a lo sumo,* $O(K(\Delta)m)$ *envíos y requiere un tiempo* $O(K(\Delta)m)$.

Demostración. En la primera fase se utiliza el algoritmo de caminos incrementales mínimos de capacidad Δ y se detiene la ejecución de este cuando $d(s) \geq K(\Delta)$. De esta manera, el algoritmo actualiza la etiqueta distancia de un nodo a lo sumo $K(\Delta)$ veces. Esto es así ya que si un nodo tiene una distancia mayor o igual que $K(\Delta)$, este nodo no forma parte del camino admisible actual ($d(i)<d(s)<K(\Delta)$). Consecuentemente, la etiqueta distancia de cualquier nodo seleccionado en un camino parcial debe ser menor que $K(\Delta)$ y, como cada paso Retroceder incrementa, en el peor de los casos, la etiqueta distancia de un nodo en una unidad, entonces el mayor número de actualizaciones de $d(i)$ es a lo sumo $K(\Delta)$. Por tanto, el número de operaciones *Retroceder* está acotado por $O(nK(\Delta))$, lo que implica que el esfuerzo computacional para realizar dichas operaciones es $O(mK(\Delta))$.

Veamos ahora cuál es el número de envíos de flujo. Para ello diremos que un envío es *Δ-saturante* a través de (i,j) si, después de haber sido realizado, la capacidad residual del arco (i,j) es estrictamente menor que Δ. Así, en la primera fase de una Δ-fase fija de escalado, el algoritmo realiza a lo sumo $K(\Delta)/2$ envíos Δ-saturantes sobre un arco, debido a que entre dos envíos Δ-saturantes consecutivos a través de (i,j), ambas $d(i)$ y $d(j)$ deben haber incrementado en al menos dos unidades (lema 1.1) y el mayor

valor de la etiqueta distancia de un nodo es $K(\Delta)$. Así, sumando para todos los arcos, se tiene que el número de envíos Δ-saturantes es $O(K(\Delta)m)$. Esto implica que el número de caminos incrementales mínimos identificados es a lo sumo $O(K(\Delta)m)$, debido a que cada envío Δ-satura al menos un arco. Como el esfuerzo computacional de cada envío de flujo es O(1), el esfuerzo total vuelve a ser $O(K(\Delta)m)$. El número de llamadas a *Avanzar* y a *Enviar_Flujo* está acotado por el número de caminos incrementales identificados más el número de operaciones *Retroceder*, es decir, $O(K(\Delta)m + K(\Delta)n)$. Por lo tanto, la complejidad de la primera fase es $O(K(\Delta)m)$. □

Lema 2.2. *La segunda fase identifica a lo sumo* $4Un^2 / K(\Delta)^2\Delta$ *caminos incrementales y requiere un tiempo* $O(4Un^2m / K(\Delta)^2\Delta)$.

Demostración. Denotamos por $f^{K(\Delta)}$ el valor del flujo al finalizar la primera fase y por f^* el valor del flujo Δ-óptimo. A continuación probaremos que $f^* - f^{K(\Delta)} \leq 4Un^2 / K(\Delta)^2$. Para ver esto llamemos V_p al conjunto de nodos que están a una distancia p del nodo sumidero t, es decir, $V_p = \{i \in V : d(i) = p\}$ y denominemos a este conjunto el p-ésimo nivel. Supondremos que los conjuntos $V_{K(\Delta)}, V_{K(\Delta)-1}, \ldots, V_1$ son distintos del vacío, ya que si alguno fuese igual al vacío, no hay caminos incrementales de s a t en $R(\Delta)$ y, por lo tanto, $f^{K(\Delta)}$ es Δ-óptimo. Hemos de darnos cuenta de que $|V_1| + |V_2| + \ldots + |V_{K(\Delta)}| \leq n-1$, debido a que el nodo sumidero no está incluido en estos conjuntos. A continuación mostraremos que la red $R(\Delta)$ contiene al menos dos niveles consecutivos, cada uno de ellos conteniendo a lo sumo $2n / K(\Delta)$ nodos. Si esto no fuese así, todo nivel alternativo contendrá $H > 2n / K(\Delta)$ nodos y por lo tanto el número total de nodos debería ser $H \cdot (K(\Delta)/2) > n$, lo que implicaría una contradicción. De esta manera, suponemos que $|V_p| \leq 2n / k(\Delta)$ y $|V_{p-1}| \leq 2n / k(\Delta)$ para alguno de dos niveles consecutivos. En este caso un posible s-t corte viene definido por los arcos que conectan $|V_p|$ con $|V_{p-1}|$, ya que no hay arcos (i, j) en $R(\Delta)$ con $i \in V_p$ y $j \in V_z$ tal que $p > z+1$ (debido a que la función distancia es válida). La capacidad de estos arcos

está acotada por U. Entonces, la capacidad de este s-t corte es $|V_p||V_{p-1}|U \leq 4Un^2 / K(\Delta)^2$, y por lo tanto, $f^* - f^{K(\Delta)} \leq 4Un^2 / K(\Delta)^2$.

Así, en el peor de los casos, en la segunda fase del algoritmo resta por enviar $4Un^2 / K(\Delta)^2$ unidades de flujo. Pero como cada envío lleva al menos Δ unidades de flujo, se necesitan a lo sumo $4Un^2 / K(\Delta)^2\Delta$ envíos. Además, después de $4Un^2 / K(\Delta)^2\Delta$ envíos de flujo se obtiene un flujo *Δ-óptimo*. Por todo esto, el esfuerzo computacional de la segunda fase es, como mucho, $4Un^2m / K(\Delta)^2\Delta$, ya que identificar un camino incremental mediante un recorrido en profundidad requiere un esfuerzo O(m). □

Resumiendo, en una Δ-fase de escalado fija, la primera fase del algoritmo emplea un tiempo O($K(\Delta)m$) y la segunda fase requiere un tiempo O($4Un^2m / K(\Delta)^2\Delta$). Es decir, un tiempo total O($K(\Delta)m + 4Un^2m / K(\Delta)^2\Delta$). Esta función alcanza un valor mínimo para $K(\Delta) = 2\left(Un^2 / \Delta\right)^{1/3}$, obteniendo un algoritmo de tiempo $O\left(\left(Un^2 / \Delta\right)^{1/3} m\right)$ para cada Δ-fase de escalado fija. El siguiente teorema establece la complejidad del algoritmo.

Teorema 2.1. *El algoritmo de dos fases escalado para* $K(\Delta) = \min(n, 2\left(Un^2 / \Delta\right)^{1/3})$ *requiere un tiempo* $O(nm \log(\frac{U}{n}))$.

Demostración. El algoritmo realiza $q = \lfloor \log U \rfloor + 1$ fases de escalado. Hemos de notar que $U = 2^{q-1}$ y que $\Delta = 2^{q-i}$ con i=1,...,q. Por lo tanto la complejidad del algoritmo viene dada por la siguiente suma:

$$\sum_{i=1}^{q} \left(2^{q-1} n^2 / 2^{q-i}\right)^{1/3} m = n^{2/3} m \sum_{i=1}^{q} (2^{1/3})^{i-1}$$

Pero, si $K(\Delta)> n$, la primera fase del algoritmo termina cuando $d(s)\geq n$, obteniendo un flujo Δ-óptimo, pues no hay caminos incrementales. Por lo tanto, supongamos que p es la fase a partir de la que $K(\Delta) \geq n$. Entonces, a partir de esta fase, cada una de ellas requiere un tiempo O(nm). A continuación calculamos el valor de p resolviendo la siguiente igualdad:

$$K(\Delta) = 2\left(Un^2/\Delta\right)^{1/3} = 2\left(2^{q-1}n^2/2^{q-p}\right)^{1/3} = 2\left(2^{p-1}n^2\right)^{1/3} = n$$

y obtenemos que $p = \log n - 2$.

A partir de la fase de escalado p+1 hasta la fase de escalado q, $K(\Delta)$ es mayor que n. Entonces la complejidad del algoritmo viene dada por la siguiente suma:

$$n^{2/3}m\sum_{i=1}^{p}(2^{1/3})^{i-1} + \sum_{p+1}^{q} mn =$$

$$n^{2/3}m\left(\frac{\left(2^{1/3}\right)^p - 1}{2^{1/3}-1}\right) + mn\,(q-p) \leq n^{2/3}m\left(2^{1/3}\right)^p + mn\,(\log U - \log n + 2)$$

Como $\left(2^{1/3}\right)^p \leq n^{1/3}$, obtenemos que la complejidad teórica del algoritmo es O($nm(\log(\frac{U}{n}))$).□

3.4 Mejoras del algoritmo en la práctica

A continuación introduciremos estrategias para mejorar el comportamiento práctico del algoritmo. Estas estrategias tienden a mejorar el algoritmo de caminos incrementales mínimos utilizado en la primera fase.

En el algoritmo de caminos incrementales mínimos puede ocurrir que no hayan caminos incrementales antes de que la etiqueta distancia del nodo fuente sea mayor o igual que $K(\Delta)$. Esto significa que el flujo actual es un flujo Δ-óptimo y, por tanto, que la actual Δ-fase de escalado debe concluir. Si no hay camino incremental en $R(\Delta)$, esto implica que debe haber algún nivel vacío.

Es decir, no hay nodos en $R(\Delta)$ cuya etiqueta distancia sea l, tal que $1 \leq l \leq K(\Delta)$. Para detectar esta situación, Ahuja y Orlin [5] proponen mantener un vector n-dimensional que llamaremos *Nnodos* , cuyos índices varían desde 0 hasta n-1. El valor de *Nnodos*(i) es el número de nodos que están a distancia i del nodo t. El computo inicial de este vector se realiza cuando en el algoritmo son calculadas las etiquetas distancias exactas. En un paso *Retroceder* sobre un nodo j, siempre se resta una unidad de *Nnodos*($d(j)$) y se examina si *Nnodos*($d(j)$)=0. Si esto último ocurre, entonces la Δ-fase de escalado ha concluido y se continúa con la siguiente. Esta estrategia es similar a la comentada en la sección (2.1.1), denominada estrategia de Derigs y Meier.

La segunda estrategia, que proponemos tiene que ver con la máxima cantidad de flujo que se puede enviar en la segunda fase. En dicha fase se pueden enviar, a lo sumo, $4Un^2 / K(\Delta)^2$ unidades de flujo. Por lo tanto, si en la primera fase detectamos un s-t corte tal que la capacidad del mismo es inferior o igual a $4Un^2 / K(\Delta)^2$, esta termina y el algoritmo continúa con la segunda fase. Para detectar este corte, proponemos el uso de un vector n-dimensional que llamaremos *Ccorte*, cuyos índices también varían desde 0 hasta n-1. El valor de *Ccorte*(i) es la suma de las capacidades residuales de los arcos que están a distancia i del nodo t. La inicialización de este vector se lleva a cabo en el cálculo de las etiquetas distancias exactas iniciales. Cada vez que se realiza un paso *Retroceder* en un nodo j, se restan $\sum_{u \in Suc(j)} r_{ju}$ unidades a *Ccorte*($d(j)$) y se examina si *Ccorte*($d(j)$)$\leq$ $4Un^2 / K(\Delta)^2$. Si esto ocurre, la primera fase termina y se continúa con la segunda. Se puede observar que esta estrategia no empeora la complejidad del algoritmo ya que el cómputo de $\sum_{u \in Suc(j)} r_{ju}$ se realiza en el paso *Retroceder* y ambas operaciones necesitan recorrer la lista de adyacencia del nodo al que se le actualiza la etiqueta distancia. Por tanto se pueden simultanear.

Debemos comentar que, en el caso de que la segunda estrategia ocurra, puede suceder que la primera fase acabe con un camino incremental parcial en el que se han ido modificando las capacidades residuales de dicho camino. Por lo tanto es necesario, antes de pasar a la segunda fase, restablecer las capacidades de los

arcos en ese camino parcial. Para ello, utilizando el vector de predecesores, retrocedemos hasta el nodo fuente, y llamamos al procedimiento *Enviar_Flujo*($Pred(i), i, -\Delta$) para cada arco del camino parcial.

Ambas estrategias se pueden incorporar al algoritmo. Si la primera estrategia tiene lugar, se pasa a la siguiente Δ-fase de escalado. Si es la segunda estrategia la que tiene lugar, se pasa a la segunda fase del algoritmo. Si ambas estrategias ocurren a la vez, evidentemente se pasa a la siguiente fase de escalado. Para facilitar posteriores referencias a estas estrategias, a la primera estrategia la codificaremos por PC (parada por corte) y a la segunda por PF (parada por fin de la primera fase).

3.5 Resultados experimentales

En este apartado reflejaremos los resultados obtenidos en la implementación de los algoritmos mencionados con anterioridad en un reducido experimento. Dejamos para el final de este capítulo, un experimento más ambicioso donde se incluye una generalización del algoritmo introducido en esta sección. En este estudio, hemos implementado los siguientes algoritmos: algoritmo de dos fases escalado en las capacidades con la estrategia PC que codificaremos por 2FEC, algoritmo de dos fases escalado en las capacidades con las estrategias PC y PF que codificaremos por 2FEC+, el algoritmo de caminos incrementales mínimos incorporando la estrategia PC y que codificaremos por CIM y el algoritmo de escalado en la capacidades de Ahuja y Orlin [5], que utiliza también la estrategia PC y que codificaremos por EC.

Todos los algoritmos han sido codificados en Pascal estándar y ejecutados en una estación de trabajo 715/33 HP9000.

Hemos seleccionado, como variables a medir, el tiempo de CPU en segundos tomado por cada algoritmo para resolver el problema. Este tiempo no incluye el tiempo empleado en los procedimientos de entrada/salida. También hemos contabilizado, para cada algoritmo, el número de envíos realizados (*ENVIOS*) y el número de pasos *Retroceder* (*RET*).

Para obtener distintas redes aleatorias de diferentes tamaños, hemos utilizado el generador NETGEN desarrollado por Klingman et al. [52] y el generador RMFGEN dado por Goldfarb y Grigoriadis [40].

Como ya hemos comentado, en el generador NETGEN la red es especificada mediante los siguientes cuatro parámetros: *semilla*, n, m y U.

En la Tabla 2.14 se muestran los valores utilizados para el número de nodos (n), el número de arcos (m) y el ratio (m/n). En esta tabla se puede ver que para cada valor del número de nodos hemos elegido valores del número de arcos. Estos valores del número de arcos vienen dados por: $10n$, $20n$ y $30n$. Los valores de la máxima capacidad (U) han sido:100, 1000, 10000. Las combinaciones de todos los posibles valores de cada parámetro dan lugar a 27 casos particulares (3x3x3). Para cada una de estas posibles especificaciones de red, hemos considerado 10 réplicas, lo que nos da un total de 270 redes aleatorias. Las 10 semillas para cada una de las réplicas son: 12345678, 36581249, 23456183, 46545174, 35826749, 43657679, 378484689, 23434767, 56567897 y 78656756.

N	*m*	*m / n*
200	2000, 4000, 6000	10, 20, 30
400	4000, 8000, 12000	10, 20, 30
600	6000, 12000, 18000	10, 20, 30

Tabla 2.14. Número de nodos, de arcos y ratio m/n.

		Media CPU(s)				*Media Envíos*				*Media RET*			
n	*m*	*2FEC*	*2FEC+*	*CIM*	*EC*	*2FEC*	*2FEC+*	*CIM*	*EC*	*2FEC*	*2FEC+*	*CIM*	*EC*
200	2000	0,13	0,12	0,83	0,07	35,10	35,10	1617,23	12,97	115,37	115,37	17740,93	43,90
200	4000	0,19	0,19	1,65	0,11	78,60	78,60	3013,87	27,00	183,97	183,97	18443,27	75,87
200	6000	0,24	0,26	2,46	0,14	89,83	89,83	4259,20	29,07	159,20	159,20	18151,80	64,97
400	4000	0,25	0,24	3,23	0,13	38,37	38,37	3433,20	13,70	141,70	141,70	66893,50	46,73
400	8000	0,36	0,39	6,08	0,22	70,43	70,43	6097,07	25,67	195,20	195,20	63838,57	79,03
400	12000	0,52	0,54	8,04	0,28	94,07	94,07	7983,73	30,97	226,43	226,43	57619,57	85,90
600	6000	0,40	0,37	7,93	0,19	44,40	44,40	5687,07	16,70	227,37	227,37	154420,40	95,07
600	12000	0,52	0,60	15,65	0,37	57,70	57,70	10204,23	19,63	106,93	106,93	152183,83	37,87
600	18000	0,80	0,85	22,38	0,46	99,60	99,60	14028,03	36,03	247,20	247,20	149797,73	106,13
TOTAL		3,41	3,55	68,26	1,98	608,10	608,10	56323,63	211,73	1603,37	1603,37	699089,60	635,47

Tabla 2.15. Media del tiempo de CPU, Envíos y RET para el generador NETGEN.

En la Tabla 2.15, se muestran la media del tiempo de CPU de cada uno de los algoritmos así como la media del número de envíos realizados y la media del número de pasos *Retroceder*. Esta tabla ha sido construida al promediar la ejecución de los algoritmos en

las réplicas y en la capacidad. Es por esto por lo que en la tabla únicamente aparecen el número de nodos y el número de arcos.

En el generador RMFGEN la red es especificada mediante los siguientes parámetros: a, b, c_1 y c_2. También, en este caso, las capacidades de los arcos están uniformemente distribuidas en el intervalo [c_1, c_2] (sólo para los arcos entre niveles, ver Goldfard y Grigoriadis [40]). En nuestro experimento, el valor de c_1 es igual a uno y el valor de c_2 es igual a *U*. Por lo tanto, el intervalo es [1, *U*]. Para este generador hemos elegido los mismos valores de *U* que en el generador NETGEN. Como el generador RMFGEN también necesita una semilla, se reduce la especificación de los parámetros a los siguientes cuatro: *semilla*, *a*, *b* y *U*.

Los valores de *a* y *b* han sido elegidos en el conjunto {4, 8, 12}. Por ello, las combinaciones de todos los posibles valores de cada parámetro dan lugar a 27 casos particulares (3x3x3). Para cada una de estas posibles especificaciones de red, hemos considerado 10 réplicas utilizando las mismas semillas que en el generador anterior, lo que nos da un total de 270 redes aleatorias.

En la Tabla 2.16, se muestran la media del tiempo de CPU, la media del número de envíos y la media del número de pasos *Retroceder* realizados por cada algoritmo. De nuevo esta tabla ha sido construida al promediar la ejecución de los algoritmos en las réplicas y en la capacidad. Es por esto por lo que en la tabla aparecen únicamente los parámetros *a* y *b*.

		Media CPU (s)				*Media Envíos*				*Media RET*			
a	*b*	*2FEC*	*2FEC+*	*CIM*	*EC*	*2FEC*	*2FEC+*	*CIM*	*EC*	*2FEC*	*2FEC+*	*CIM*	*EC*
4	4	0,05	0,04	0,01	0,03	77,07	77,07	39,07	28,00	1869,33	1869,33	306,83	970,17
4	8	0,09	0,09	0,02	0,07	75,70	75,70	75,63	39,70	4430,97	4430,97	893,30	3041,10
4	12	0,15	0,14	0,04	0,10	75,80	75,80	107,40	46,90	7984,00	7980,57	1709,80	5943,73
8	4	0,34	0,34	0,06	0,22	306,83	306,83	175,13	130,33	23842,37	23842,37	3117,60	16762,57
8	8	0,77	0,76	0,15	0,54	309,60	309,60	369,00	186,40	54789,63	54789,63	8596,77	46420,70
8	12	1,33	1,30	0,26	0,94	306,60	306,60	543,03	215,70	95964,83	95952,70	16270,13	86402,30
12	4	1,25	1,23	0,18	0,81	696,60	696,60	407,37	308,87	94101,23	94101,23	11542,40	74890,70
12	8	2,85	2,79	0,46	1,98	693,07	693,07	873,27	434,93	212572,93	212572,93	30118,67	190269,83
12	12	4,72	4,61	0,81	3,38	688,77	688,77	1304,77	501,30	350738,50	350738,50	54276,97	330235,40
TOTAL		11,55	11,31	2,00	8,06	3230,03	3230,03	3894,67	1892,13	846293,80	846278,23	126832,47	754936,50

Tabla 2.16. Media del tiempo de CPU, Envíos y RET para el generador RMFGEN.

Sobre las Tablas 2.15 y 2.16 pueden ser realizadas las siguientes puntualizaciones:

Para las redes generadas por NETGEN, resulta más rápido el algoritmo EC y, a continuación, el algoritmo 2FEC. Además, la

estrategia PF no mejora el tiempo de CPU del algoritmo de dos fases escalado. Sin embargo, los tiempos para EC, 2FEC y 2FEC+ son muy inferiores al tiempo de CIM. Ocurre lo mismo en cuanto a la media del número de envíos y de llamadas a *Retroceder* (RET). A destacar que, para 2FEC y 2FEC+, el número de envíos y de RET es exactamente el mismo, lo que implica que la estrategia PF no ha tenido lugar.

Para las redes generadas por RMFGEN, las cosas cambian. En este caso, el más rápido es CIM, a pesar de realizar el mayor número de envíos. Además, este algoritmo es el de menor valor de RET. El que este algoritmo sea más rápido para este generador es debido a que las redes que genera RMFGEN son por niveles y el corte mínimo se encuentra entre dos niveles consecutivos, lo que beneficia a CIM. El siguiente algoritmo es, otra vez, EC y, por último, y por este orden, 2FEC+ y 2FEC. De nuevo, para 2FEC+ no ha tenido lugar la estrategia PF.

En ambos, generadores el algoritmo EC realiza menor número de envíos que 2FEC y 2FEC+. Esto es así ya que, en una fase Δ, EC envía por cada camino incremental mínimo una cantidad de flujo igual a la capacidad residual del camino que es mayor o igual a Δ y menor estricto que 2Δ. En cambio 2FEC y 2FEC+ siempre envían Δ unidades de flujo por cada camino incremental en la primera fase. Por esto, el número de envíos realizado por EC es menor, aunque es semejante al que realiza el algoritmo de dos fases escalado si lo comparamos con el grandísimo número que realiza CIM.

4. Algoritmo de dos fases doblemente escalado en las capacidades

En esta sección proponemos un algoritmo de dos fases doblemente escalado en las capacidades que generaliza el introducido en la sección anterior. Este algoritmo requiere un tiempo $O(nm \log\beta \log_\beta(U/n))$ = $O(nm \log(U/n))$ para $\beta \leq n/8$ y un tiempo $O(nm \log\beta \log_\beta U)$ = $O(nm \log U)$ para $\beta > n/8$. Se observa que la complejidad del algoritmo puede hacerse depender del parámetro β que define la base de la primera escala en las capacidades.

Para definir la primera escala en las capacidades introducimos un parámetro Δ_β y nos referimos a la red Δ_β-*residual*, $R(\Delta_\beta)$, como la red que únicamente contiene arcos cuya capacidad residual es al menos Δ_β. De esta manera, todos los caminos incrementales en $R(\Delta_\beta)$ tienen una capacidad residual de al menos Δ_β. Se dice que un flujo es Δ_β-*óptimo* si en R no existe ningún camino incremental con capacidad residual de al menos Δ_β. El algoritmo trabaja en fases de escalado; cada fase tiene un Δ_β fijo y, cuando finaliza una fase, comienza la siguiente con $\Delta_\beta = \Delta_\beta / \beta$. Al principio $\Delta_\beta = \beta^{\lceil \log_\beta U \rceil}$; es decir, es igual al menor entero potencia de β que es mayor o igual que U. La última fase del algoritmo ocurre cuando $\Delta_\beta = 1$. En esta última fase $R(\Delta_\beta) = R$ y, por tanto, al finalizar el algoritmo se determina un flujo máximo. De esta manera, el algoritmo realiza $|\log_\beta U|$ fases de escalado.

En cada Δ_β-fase de escalado se llama al algoritmo de dos fases de escalado de la sección anterior. En este caso, dicho algoritmo considera la red $R'(\Delta_\beta)$. La red $R'(\Delta_\beta)$ es obtenida a partir de $R(\Delta_\beta)$ con las capacidades residuales $r'_{ij} = r_{ij} / \Delta_\beta$. Es decir, $R'(\Delta_\beta)$ tiene el mismo conjunto de nodos y de aristas que $R(\Delta_\beta)$, pero las capacidades residuales son divididas por Δ_β (división entera). Denotaremos por $U'(\Delta_\beta)$ a la mayor capacidad residual de $R'(\Delta_\beta)$. Así, $U'(\Delta_\beta) = U / \Delta_\beta$ y, para la primera Δ_β-fase de escalado, $\Delta_\beta \geq U$; es decir, $U'(\Delta_\beta) = 1$. Cuando $\Delta_\beta = 1$, entonces $U'(\Delta_\beta) = U$.

Debemos notar que, en una Δ_β-fase de escalado, no hay caminos incrementales en $R(\Delta_\beta)$ de cuello de botella mayor o igual que $\beta \cdot \Delta_\beta$. Por tanto, en $R'(\Delta_\beta)$ todos los caminos incrementales tienen una capacidad residual inferior a β. Así, el algoritmo de dos fases de escalado de la sección anterior realiza $\lceil \log \beta \rceil$ Δ-fases de escalado, comenzando en la fase $\Delta = 2^{\lceil \log \beta \rceil}$ para la red $R'(\Delta_\beta)$ con capacidad máxima $U'(\Delta_\beta)$. De esta manera, fijada una Δ_β-fase y terminada una Δ-fase, en $R'(\Delta_\beta)$ no hay caminos incrementales de

capacidad residual mayor que Δ y, por tanto, en R no hay caminos incrementales de capacidad mayor que $\Delta_\beta\Delta$. El esquema del algoritmo es el siguiente

```
Algoritmo 2FDEC; {dos fases doblemente escalado en las
capacidades}
  begin
  X:=0; Sea la red residual R(x);
  Δβ := β^⌈logβ U⌉;
  while Δβ ≥ 1 do
    begin
    2FEC( β ,U'(Δβ));
    Δβ := Δβ / β
    end
  end.
```

El algoritmo *2FDEC* llama al algoritmo *2FEC*, $\left|\log_\beta U\right|$ veces. *2FEC* es llamado con el parámetro β, ya que la primera Δ-fase de escalado $\Delta = 2^{\lceil \log \beta \rceil}$, y con $U'(\Delta_\beta)$ que denota la capacidad máxima para la red $R'(\Delta_\beta)$. El valor de $U'(\Delta_\beta)$ es necesario para computar el valor de $K(\Delta, \Delta_\beta)$. En el esquema anterior se omite la construcción de la red $R'(\Delta_\beta)$ (es decir, 2FEC trabaja con la red R directamente). Para ello, en una Δ_β-fase de escalado y una Δ-subfase de escalado, se consideran únicamente los arcos (i, j) de R tales que $r_{ij} \geq \Delta_\beta\Delta$. Así, los procedimientos *Enviar_Flujo*, *Avanzar* y *Retroceder* de la primera fase del algoritmo *2FEC* se modifican de la manera siguiente:

```
Procedure Enviar_Flujo(i,j,Δ);
  begin
  rij = rij − ΔβΔ ; rji = rji + ΔβΔ
  end;

Procedure Avanzar(i);
  begin
  Sea (i,j) un arco ΔβΔ-admisible que sale de i;
  pred(j) := i; i := j
  end;

Procedure Retroceder(i);
  begin
  d(i) := min{d(j) + 1 : (i, j) ∈ A(i) y rij > ΔβΔ};
  if i ≠ s then i := pred(i)
```

end;

En la segunda fase del algoritmo *2FEC*, el algoritmo de Ford y Fulkerson identifica caminos incrementales de cuellos de botella mayores o iguales que $\Delta_\beta \Delta$.

4.1 Complejidad del algoritmo de dos fases doblemente escalado en las capacidades

Para el cálculo de la complejidad del algoritmo de dos fases doblemente escalado en las capacidades se precisa conocer la complejidad del algoritmo de dos fases escalado en las capacidades (ver sección anterior). La complejidad de este último algoritmo depende del valor de $K(\Delta)$ y viene dada, para una Δ-fase, por O($K(\Delta)m + 4Un^2m / K(\Delta)^2\Delta$). Recordamos que el algoritmo se ejecuta, para una Δ-fase fija, en un tiempo $O\left(\left(Un^2/\Delta\right)^{1/3} m\right)$ y que el valor de $K(\Delta)$ venía dado por la expresión $K(\Delta) = \min(n, 2\left(Un^2/\Delta\right)^{1/3})$.

Para el algoritmo doblemente escalado, una vez fijada una Δ_β-fase de escalado, para la Δ-subfase de escalado se tiene que:

$$K(\Delta, \Delta_\beta) = 2\left(U'(\Delta_\beta)n^2 / \Delta\right)^{1/3}$$

Como $U'(\Delta_\beta) = U / \Delta_\beta$, se tiene que $K(\Delta, \Delta_\beta) = 2\left(Un^2 / \Delta_\beta\Delta\right)^{1/3}$.

Así, en una Δ_β-fase hay $\lceil \log \beta \rceil + 1$ Δ-subfases, y por lo tanto, el tiempo de ejecución para una Δ_β-fase de escalado viene dado por la expresión:

$$\sum_{j=0}^{\lceil \log \beta \rceil} \left(K(\Delta, \Delta_\beta)m + 4Un^2m / \Delta_\beta \Delta K(\Delta, \Delta_\beta)^2 \right) = \frac{3}{2} \sum_{j=0}^{\lceil \log \beta \rceil} \left(K(\Delta, \Delta_\beta)m \right)$$

Finalmente, el tiempo de ejecución del algoritmo *2FDEC* es la suma de la expresión anterior para todas las Δ_β-fases de escalado, es decir:

$$\sum_{i=0}^{\lceil \log_\beta U \rceil} \frac{3}{2}\left(\sum_{j=0}^{\lceil \log \beta \rceil} \left(K(\Delta, \Delta_\beta) m \right) \right)$$

Sean $q = \lfloor \log_\beta U \rfloor$ y $q' = \lceil \log \beta \rceil$. Entonces, $U = \beta^q$ y $\Delta_\beta = \beta^{q-i} = U / \beta^i$ para i=0,...,q. De la misma forma se tiene que $\beta = 2^{q'}$ y $\Delta = 2^{q'-j} = \beta / 2^j$ para j=0,..., q'. Así tenemos que $K(\Delta, \Delta_\beta) = 2\left({Un^2 \beta^i 2^j}/{U\beta} \right)^{1/3} = 2\left(\beta^{i-1} 2^j n^2 \right)^{1/3}$.

Teorema 2.2. *El algoritmo 2FDEC, para* $\beta > n/8$, *requiere un tiempo* O($nm \log U$).

Demostración. Sabemos que si $K(\Delta, \Delta_\beta) \geq n$, la Δ-subfase de escalado, fijada un Δ_β-fase de escalado, requiere un tiempo O(nm). Ahora, la cuestión es ver para qué valor de β con independencia de la fase Δ_β se tiene que $K(\Delta, \Delta_\beta) \geq n$. Para ello, se resuelve la siguiente desigualdad:

$$K(\Delta, \Delta_\beta) = 2\left(\beta^{i-1} 2^j n^2 \right)^{1/3} \geq n \Leftrightarrow \beta^{i-1} \geq {n}/{8 \cdot 2^j} \Rightarrow \beta^{i-1} \geq {n}/{8} \geq {n}/{8 \cdot 2^j}$$

Es decir, se ha de cumplir que el parámetro β ha de satisfacer $\beta^{i-1} \geq n/8$. Así, tenemos que, a partir de la Δ_β-fase de escalado tal que $i \geq \log_\beta n/8 + 1$, cada Δ-subfase de escalado corre en un tiempo O(nm). Si $\beta > n/8$ entonces, a partir de $i \geq 1$, se cumple la condición anterior. Por lo tanto, para i=1,...,q, cada Δ-subfase corre en un tiempo O(nm) con independencia de los valores de Δ_β y Δ. Luego la complejidad teórica viene dada por la siguiente suma:

$$\sum_{i=1}^{q} \sum_{j=0}^{q'} nm = nm \sum_{i=1}^{q} (q' + 1) = nm \sum_{i=1}^{q} \log \beta + 1 = nmq\ (\log \beta + 1) = nm\ \log_\beta U (\log \beta + 1)$$

Considerando únicamente los términos dominantes O($nm \log_{\beta} U \log \beta$), el tiempo de ejecución es O($nm \log U$).□

Teorema 2.3. *El algoritmo 2FDEC, para* $\beta \le n/8$ *y para* $K(\Delta,\Delta_{\beta}) = \min(n, 2\left(Un^2 / \Delta_{\beta}\Delta\right)^{1/3})$*, requiere un tiempo* O($nm \log_{\beta}(U/n) \log \beta)$) = O($nm \log(U/n)$).

Demostración. El algoritmo realiza $q = \lfloor \log_{\beta} U \rfloor$ Δ_{β}-fases de escalado. En cada Δ_{β}-fase se realizan $q' = \lceil \log \beta \rceil$ Δ-subfases de escalado. Para $\beta \le n/8$ y, fijada una Δ_{β}-fase de escalado, sea p' la Δ-subfase a partir de la cual $K(\Delta,\Delta_{\beta}) \ge n$. Para calcular p', resolvemos la desigualdad $2\left(\beta^{i-1} 2^{p'} n^2\right)^{1/3} \ge n$ y obtenemos que $p' \ge \log\left(n / 8\beta^{i-1} \right)$. Ahora la cuestión es ver cuándo $q' \ge p'$. Para responder a esta pregunta debemos obtener la Δ_{β}-fase p tal que $\log \beta \ge \log\left(n / 8\beta^{p-1} \right)$. Por lo tanto, a partir de la Δ_{β}-fase $p = \log_{\beta}(n/8)$, tenemos que $q' \ge p' \ge 0$ (por ser $\beta \le n/8$). Por lo tanto, cada Δ-subfase de escalado desde p' hasta q' tarda un tiempo O(nm), debido a que $K(\Delta,\Delta_{\beta}) \ge n$. Sin embargo, a partir de la Δ_{β}-fase p+1 hasta la Δ_{β}-fase q se tiene que $p' \le 0$, debido a que $p' = \log\left(n / 8\beta^{p+r} \right) = \log(\beta^{-r})$, con r tomando valores en el conjunto {0,..., $q-p-1$ }. Por lo tanto, a partir de la Δ_{β}-fase de escalado p+1, cada Δ-subfase de escalado desde 0 hasta q' requiere un tiempo O(nm). En resumen, la complejidad teórica del algoritmo para $\beta \le n/8$ viene dada por la siguiente suma:

$$\sum_{i=0}^{p} \frac{3}{2} \left(\sum_{j=0}^{q'} \left(K(\Delta,\Delta_{\beta}) m \right) \right) + \sum_{i=p+1}^{q} \left(\sum_{j=0}^{q'} (nm) \right)$$

El primer término de la anterior suma se refiere al esfuerzo realizado por el algoritmo en las Δ_{β}-fases de escalado con i=0,...,p

para las que $K(\Delta,\Delta_\beta) < n$. El segundo término se refiere a las Δ_β-fases de escalado con $i=p+1,\ldots,q$ en donde, para cada una de ellas, en todas las subfases de escalado se tiene que $K(\Delta,\Delta_\beta) \geq n$. A continuación realizaremos la suma de cada término anterior por separado. El primer término es:

$$\sum_{i=0}^{p} \frac{3}{2}\left(\sum_{j=0}^{q'}\left(K(\Delta,\Delta_\beta)m\right)\right) = \sum_{i=0}^{p}\sum_{j=0}^{q'} 3(\beta^{i-1}2^j n^2)^{1/3} m = \sum_{i=0}^{p} 3n^{2/3} m(\beta^{1/3})^{i-1} \frac{(2^{1/3})^{q'+1}-1}{2^{1/3}-1}$$

$$= \sum_{i=0}^{p} 3n^{2/3} m(\beta^{1/3})^{i-1} \frac{\beta^{1/3}2^{1/3}-1}{2^{1/3}-1} = 3n^{2/3} m \frac{(\beta^{1/3})^p - 1/\beta^{1/3}}{\beta^{1/3}-1} \frac{\beta^{1/3}2^{1/3}-1}{2^{1/3}-1}$$

$$\leq 3n^{2/3} m \frac{(\beta^{1/3})^{p+1}-1}{\beta^{1/3}-1} \frac{2^{1/3}}{(2^{1/3}-1)}$$

En la anterior suma se debe considerar que $\beta \geq 2$ y, por lo tanto, $(2^{1/3})^{q'} = \beta^{1/3} > 1$. Recordando que $\left(\beta^{1/3}\right)^p = n^{1/3}/2$ y quedándonos con los términos dominantes, tenemos que la anterior expresión se puede acotar por $O(nm)$.

Finalmente, el segundo término de la suma es:

$$\sum_{i=p+1}^{q}\left(\sum_{j=0}^{q'}(nm)\right) = \sum_{i=p+1}^{q} nm(q'+1) = \sum_{i=p+1}^{q} nm(\log\beta + 1)) = nm(\log\beta+1)(q-p)$$

$$= nm(\log\beta+1)(\log_\beta U - \log_\beta(n/8)) = nm(\log\beta+1)\log_\beta(8U/n)$$

Resumiendo, la complejidad del algoritmo tiene la expresión $O(nm\log_\beta(U/n)\log\beta) = O(nm\log(U/n))$. □

La Figura 2.2 resume la complejidad para el algoritmo doblemente escalado de dos fases (2FDEC). En la Figura 2.2a) se puede observar que las primeras $\log_\beta(n/8)$ Δ_β-fases de escalado requieren un esfuerzo computacional de $O(nm)$. Cada una de las restantes fases de escalado precisa un esfuerzo de $O(nm\log\beta)$. La Figura 2.2b) muestra el mismo esquema pero para el algoritmo de escala en las capacidades de dos fases (2FEC). En este caso las primeras $\log(n/8)$ Δ-fases de escalado requieren un esfuerzo de

O(nm) y cada una de las restantes corre en un tiempo de O(nm). Ambos algoritmos coinciden para $\beta = 2$. Se puede observar en la Figura 2.2a) que, para $\beta > n/8$, todas las Δ_β-fases de escalado necesitan un tiempo O($nm \log \beta$), obteniendo un algoritmo de complejidad O($nm \log U$).

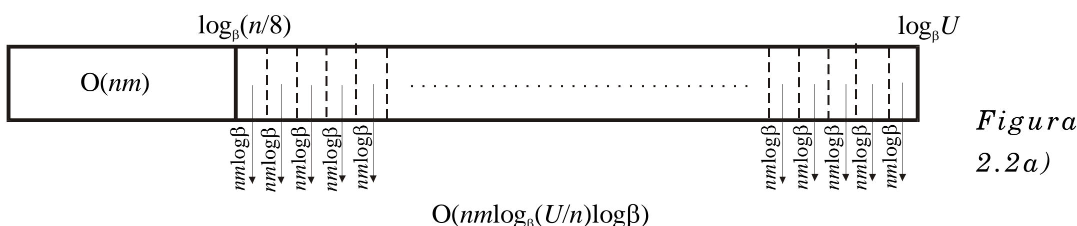


Figura 2.2a)

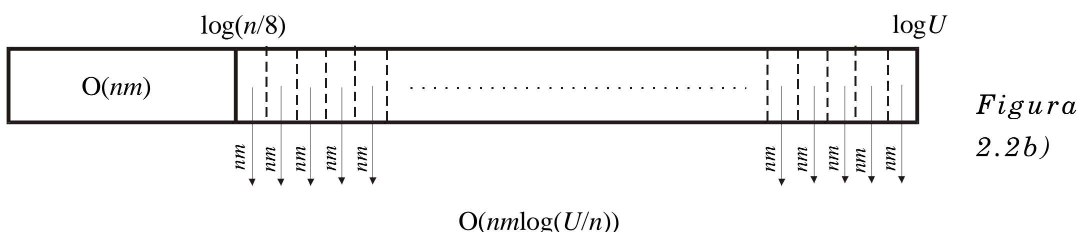


Figura 2.2b)

Figura 2.2. Comparativa de la complejidad de 2FEC y 2FDEC.

Finalmente, podemos comentar que la consideración de una doble escala en las capacidades no mejora desde un punto de vista teórico la complejidad del algoritmo. Como acabamos de ver, en el caso peor, el algoritmo doblemente escalado y el que considera una única escala corren en un tiempo O($nm \log(U/n)$). Sin embargo es lógico cuestionar si en la práctica se da la misma situación. Para ello, proponemos el siguiente estudio computacional.

4.2 Estudio computacional

En este apartado presentamos un estudio computacional para el algoritmo 2FDEC, con β tomando los valores {2,3,4,5,6,7,8,9}, con la intención de explicar el comportamiento empírico del algoritmo para los distintos valores de β. Además de medir el tiempo de CPU, consideramos el recuento de las operaciones representativas realizadas por el algoritmo.

El algoritmo ha sido codificado en Pascal estándar y ejecutado en una estación de trabajo 712/80 HP9000. En la implementación

del algoritmo se ha introducido la estrategia PC (parada por corte comentada en la sección anterior) que impide la realización de un gran número de pasos Retroceder después de que un flujo $\Delta_\beta\Delta$-óptimo es obtenido. No se ha incluido la estrategia PF.

De nuevo hemos utilizado en este estudio los generadores NETGEN y MFGEN. Como ya ha sido comentado, la especificación de ambos generadores queda determinada por los siguientes cuatro parámetros: *semilla*, n, m y U.

No hemos utilizado el generador RMFGEN dado por Goldfard y Grigoriadis [40] ya que no permite la generación de redes con un número de arcos superior a $6n$. En este caso, si que estamos interesados en un estudio computacional de mayor profundidad, por lo que necesitamos trabajar también con redes densas.

En la Tabla 2.17 se muestran los valores utilizados para el número de nodos (n), el número de arcos (m) y el ratio (m/n). En esta tabla se puede ver que, para cada valor del número de nodos, hemos elegido los valores del número de arcos. Estos valores vienen dados por: $10n$, $20n$, $30n$ y $40n$. Los valores de la máxima capacidad (U) han sido: 100, 10000, 1000000. Las combinaciones de todos los posibles valores de cada parámetro dan lugar a 48 casos particulares (4*4*3). Para cada una de estas posibles especificaciones de red, hemos considerado 10 réplicas, lo que nos da un total de 480 redes aleatorias. Las 10 semillas para cada una de las réplicas son: 12345678, 36581249, 23456183, 46545174, 35826749, 43657679, 378484689, 23434767, 56563897 y 78656756. Cada una de estas especificaciones han sido la entrada de los dos generadores. Por lo tanto se han resuelto un total de 960 problemas para cada valor de β.

n	m	m/n
250	2500, 5000, 7500, 10000	10, 20, 30,40
500	5000, 10000,15000 20000	10, 20, 30,40
750	7500, 15000, 22500,30000	10, 20, 30,40
1000	10000, 20000, 30000, 40000	10, 20, 30,40

Tabla 2.17. Número de nodos, de arcos y ratio m/n.

El tiempo de CPU en segundos, tomado por el algoritmo para resolver el problema, ha sido la variable seleccionada. Este tiempo no incluye el tiempo empleado en los procedimientos de entrada/salida. Con el fin de determinar las operaciones

representativas del algoritmo de dos fases doblemente escalado en las capacidades, hemos medido el tiempo total de CPU (CPU_FF) empleado por el algoritmo de Ford y Fulkerson. Por lo tanto, el tiempo total empleado por el algoritmo de caminos incrementales mínimos (fase 1) viene dado por CPU-CPU_FF. Para los problemas generados mediante NETGEN, el tiempo de CPU_FF es inferior al 5% del tiempo de CPU total, mientras que para los problemas generados mediante MFGEN no llega al 1.5%. Estas últimas afirmaciones pueden ser obtenidas a partir de los resultados presentados en la Tabla 2.18. Por lo tanto, a la hora de determinar las operaciones representativas del algoritmo, únicamente tendremos en cuenta las que determinan el esfuerzo computacional de la primera fase del algoritmo. Para el caso de $\beta \leq n/8$ las mencionadas operaciones son:

AR: El número de arcos examinados en las llamadas a *Retroceder*, es decir, el esfuerzo computacional en la actualización de las etiquetas distancias. En cada Δ-subfase de escalado, el algoritmo examina O($K(\Delta,\Delta_\beta)m$) arcos. Luego, el número total de arcos examinados esta acotado por O($nm \log(u/n)$).

AA: El número de arcos examinados para la determinación de los caminos admisibles, es decir, el esfuerzo computacional total en los envíos de flujo. El esfuerzo total en esta operación vuelve a estar acotado por O($nm \log(u/n)$).

Así, en la Tabla 2.18 se muestran los resultados experimentales para los problemas obtenidos mediante los dos generadores mencionados, dependiendo de la especificación de los valores de número de nodos (n) y el valor de β seleccionado. En dicha tabla aparece el tiempo medio de CPU, el tiempo medio de CPU_FF, el número medio de arcos examinados en los pasos *Retroceder* (*AR*) y el número medio de arcos examinados en la determinación de los caminos admisibles. Esta tabla ha sido obtenida al promediar la ejecución de los algoritmos en las réplicas, en el número de arcos (m) y en la capacidad (U). Por esto, únicamente aparecen en la tabla el número de nodos y el valor de β.

En la Tabla 2.18 se puede observar que el tiempo de CPU, para los problemas generados por NETGEN, es mayor que para los

problemas generados por MFGEN. En la Figura 2.3 se muestra el tiempo medio de CPU del algoritmo de dos fases doblemente escalado en las capacidades frente a la variación de β para los problemas obtenidos por ambos generadores. Se puede observar que el tiempo de CPU para los problemas que genera NETGEN es siempre mayor que para los correspondientes de MFGEN. Además, el comportamiento del tiempo de CPU frente a β es decreciente. En la mencionada figura se puede observar que la velocidad de decrecimiento, en función de β, está acotada superiormente por $2\beta^{-0.3}$ (con coeficiente de determinación $R^2 = 0.95$).

		NETGEN				**MFGEN**			
n	*β*	*CPU_FF*	*CPU*	*AR*	*AA*	*CPU_FF*	*CPU*	*AR*	*AA*
250	2	0,02	0,66	16122,90	20245,73	0,00	0,56	12845,11	18275,23
250	3	0,02	0,53	14139,33	17920,45	0,00	0,46	11859,66	16849,77
250	4	0,04	0,49	16088,94	20207,23	0,02	0,43	12845,11	18275,23
250	5	0,02	0,45	15353,10	19254,78	0,00	0,40	12421,18	17556,89
250	6	0,02	0,41	14937,56	18903,47	0,00	0,36	12375,18	17577,74
250	7	0,01	0,38	15308,02	19375,47	0,00	0,34	12743,17	18049,36
250	8	0,03	0,40	16098,66	20197,41	0,01	0,36	12845,11	18275,23
250	9	0,02	0,39	14945,05	18919,51	0,00	0,36	12505,46	17738,42
500	2	0,04	1,36	16844,53	21775,73	0,00	1,16	24919,62	30140,29
500	3	0,04	1,10	14005,95	18478,68	0,00	0,96	21594,63	26431,64
500	4	0,08	1,04	16843,21	21771,52	0,06	0,88	24919,62	30140,29
500	5	0,04	0,94	14414,11	18970,29	0,00	0,82	22165,77	27007,73
500	6	0,04	0,85	15006,73	19583,05	0,00	0,76	23039,98	28040,78
500	7	0,03	0,80	16077,77	20918,90	0,00	0,71	23740,61	28778,66
500	8	0,06	0,84	16586,43	21494,91	0,04	0,72	24919,62	30140,29
500	9	0,04	0,84	14651,45	19341,31	0,01	0,72	22676,82	27685,47
750	2	0,06	1,97	28849,65	34189,67	0,00	1,72	25668,55	33110,03
750	3	0,06	1,62	25197,64	30090,18	0,00	1,44	19727,63	26320,28
750	4	0,13	1,50	28784,93	34112,06	0,06	1,33	25668,55	33110,03
750	5	0,05	1,36	25406,31	30346,23	0,00	1,25	20110,45	26827,29
750	6	0,05	1,23	26380,31	31531,60	0,00	1,16	22479,78	29415,64
750	7	0,04	1,18	26869,77	31980,19	0,00	1,08	22191,32	29141,01
750	8	0,11	1,25	28885,90	34168,23	0,04	1,14	25668,55	33110,03
750	9	0,06	1,22	26629,74	31751,30	0,00	1,12	20987,33	27940,11
1000	2	0,09	2,61	24810,88	30229,49	0,00	2,31	29219,48	35213,58
1000	3	0,08	2,11	21564,91	26494,87	0,00	1,95	24222,54	29768,12
1000	4	0,18	2,00	24810,88	30229,49	0,14	1,82	29219,48	35213,58
1000	5	0,08	1,81	21424,53	26395,98	0,00	1,73	24744,59	30207,15
1000	6	0,06	1,62	22970,44	28094,33	0,00	1,58	25707,27	31442,23
1000	7	0,07	1,59	23161,34	28533,11	0,00	1,48	26441,56	32105,11
1000	8	0,14	1,64	24801,63	30160,42	0,08	1,52	29219,48	35213,58
1000	9	0,08	1,58	22906,08	28150,39	0,01	1,49	25626,74	31386,16

Tabla 2.18. Media del tiempo de CPU, AA y AR para los generadores NETGEN y MFGEN.

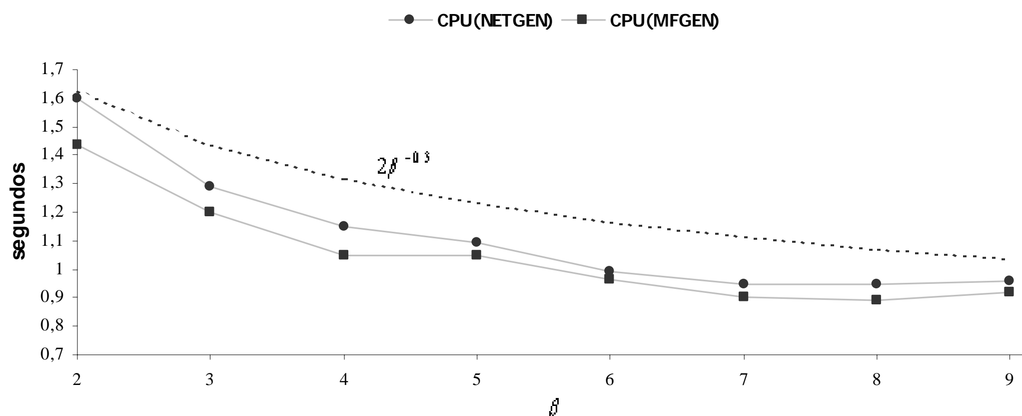


Figura 2.3. Tiempo de CPU medio frente la factor de escala β

La Figura 2.4, muestra el ratio $AA/(AA+AR)$ frente al crecimiento del factor de escala para ambos generadores. En esta figura se puede observar que el número de arcos examinados para la identificación de caminos admisibles es siempre sensiblemente superior al número de arcos examinados en la actualización de las etiquetas distancias. Así, podemos decir que, con independencia del generador, aproximadamente el 56,5% de las operaciones en las que se examinan arcos, se realizan en la detección de caminos admisibles y el resto en las actualizaciones de las etiquetas distancias. Además esta proporción permanece constante frente a los valores de β. Esto último es congruente con la teoría ya que el número de arcos examinados en las dos operaciones representativas depende de igual manera con respecto a β. Sin embargo, para el caso del algoritmo de escalado en las capacidades dado por Ahuja y Orlin [5], el numero de arcos examinados para la admisibilidad aumenta y el número de arcos examinados para la actualización de las etiquetas decrece cuando aumenta β (ver Ahuja et al. [2]).

A continuación, procedemos a estimar el *tiempo de CPU virtual* del algoritmo para los distintos valores de β, con respecto a cada generador, como una función lineal de las operaciones representativas. Esto permitirá obtener una estimación del ratio de crecimiento del tiempo de CPU como una función polinomial de los nodos (n) y arcos (m). El concepto de tiempo de CPU virtual, *CPUV*, como función de las operaciones representativas de un algoritmo es debido a Ahuja et al. [4], y fue comentado en el capítulo 1. Para el algoritmo que presentamos, el tiempo de ejecución del algoritmo viene dado por $CPUV=\alpha_{AA}AA+\alpha_{AR}AR$. Para estimar las constantes α_{AA}

y α_{AR} se realiza una regresión lineal para el tiempo de CPU (no incluye el tiempo de CPU_FF) frente a las variables independientes *AA* y *AR* para los distintos valores de β y para ambos generadores. La Tabla 2.19, muestra los valores obtenidos para las constantes del modelo, así como el valor del coeficiente de determinación R^2 para cada uno de ellos.

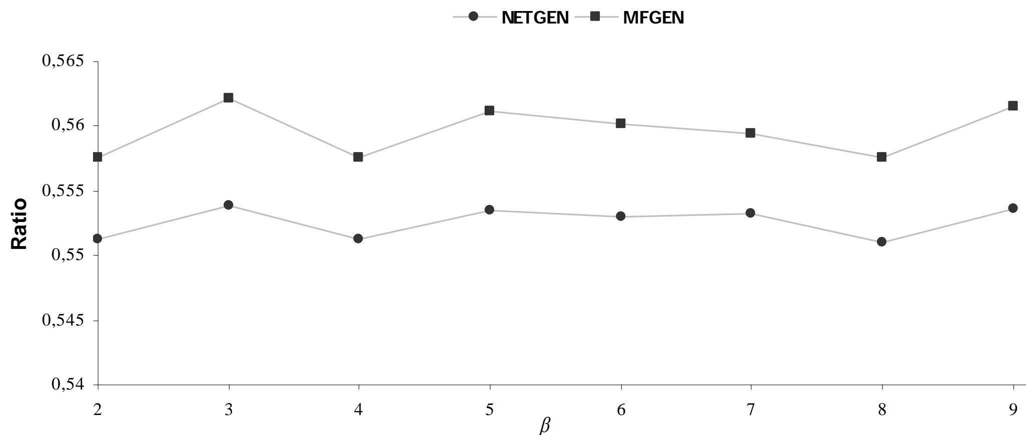


Figura 2.4. Ratio AA/(AA+AR) frente a β.

En la Tabla 2.19, se observa que la expresión lineal del tiempo de CPU virtual (CPUV) para los problemas generadores por NETGEN únicamente depende del número de arcos examinados en la detección de admisibilidad, es decir, del término *AA*. Sin embargo, no ocurre lo mismo para los problemas generados mediante MFGEN. Para ellos, cuando β es igual a 2 y a 7, depende exclusivamente de *AR*; en otro caso, depende de *AA*.

	NETGEN			**MFGEN**		
β	α_{AA}	α_{AR}	R^2	α_{AA}	α_{AR}	R^2
2	5,519e-5	0	0,877	0	5,870e-5	0,918
3	4,968e-5	0	0,874	4,474e-5	0	0,893
4	3,923e-5	0	0,866	3,388e-5	0	0,905
5	4,153e-5	0	0,866	3,850e-5	0	0,888
6	3,610e-5	0	0,873	3,397e-5	0	0,898
7	3,436e-5	0	0,876	0	3,939e-5	0,900
8	3,241e-5	0	0,869	2,879e-5	0	0,901
9	3,512e-5	0	0,876	3,277e-5	0	0,900

Tabla 2.19. Constantes para el tiempo de CPU virtual para los distintos valores de β.

A continuación calculamos el ratio de crecimiento de las operaciones representativas que influyen en el tiempo de CPU virtual como una función polinomial del tamaño de la red. Esto permitirá estimar el tiempo de CPU virtual como una función del

tamaño del problema. Supondremos que el modelo de regresión para AA y AR es, en ambos casos, $cn^{\alpha}d^{\gamma}$, donde $d = m/n$. Las constantes c, α y γ son estimadas por regresión lineal después de tomar logaritmos de AA y AR.

La Tabla 2.20, muestra la estimación de AA o AR, según aparezca ó no en el modelo lineal para el tiempo de CPU virtual (ver Tabla 2.19) para los distintos valores de β y los dos generadores.

	NETGEN		*MFGEN*	
β	*Estimación de AA ó AR*	R^2	*Estimación de AA ó AR*	R^2
2	$AA = 28{,}91n^{0{,}33}d^{1{,}45}$	0,917	$AR = 6{,}16n^{0{,}58}d^{1{,}39}$	0,873
3	$AA = 15{,}38n^{0{,}34}d^{1{,}56}$	0,942	$AA = 20{,}41n^{0{,}35}d^{1{,}49}$	0,972
4	$AA = 28{,}71n^{0{,}33}d^{1{,}45}$	0,917	$AA = 16{,}22n^{0{,}48}d^{1{,}36}$	0,904
5	$AA = 194{,}09d^{1{,}46}$	0,896	$AA = 31{,}12n^{0{,}31}d^{1{,}45}$	0,968
6	$AA = 20{,}51n^{0{,}34}d^{1{,}51}$	0,905	$AA = 28{,}64n^{0{,}36}d^{1{,}38}$	0,939
7	$AA = 22{,}28n^{0{,}34}d^{1{,}45}$	0,931	$AR = 12{,}74n^{0{,}42}d^{1{,}44}$	0,938
8	$AA = 28{,}38n^{0{,}32}d^{1{,}46}$	0,915	$AA = 16{,}22n^{0{,}48}d^{1{,}36}$	0,904
9	$AA = 19{,}36n^{0{,}34}d^{1{,}51}$	0,925	$AA = 25{,}47n^{0{,}34}d^{1{,}45}$	0,970

Tabla 2.20. Estimación de AA ó AR para los distintos valores de β.

A partir de las funciones obtenidas en la Tabla 2.20, se puede determinar el tiempo de CPU virtual sin más que multiplicar por la correspondiente constante dada en la Tabla 2.19. Así, en las Figuras 2.5 y 2.6 mostramos el ratio *CPUV/CPU* con el fin de observar la bondad de la estimación. Para ambos generadores, se puede observar que, cuando aumenta el producto *nd*, el cociente calculado se encuentra entre 0,8 y 1,1

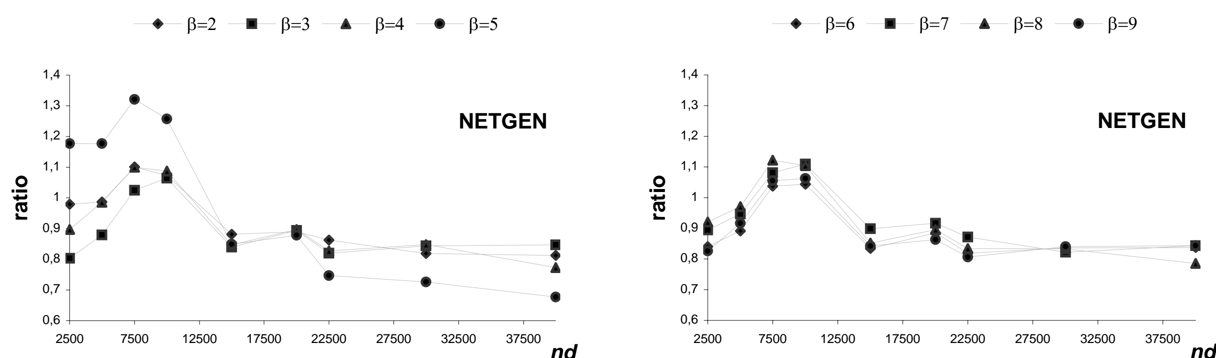


Figura 2.5. Ratio CPUV/CPU frente a nd para problemas generados con NETGEN.

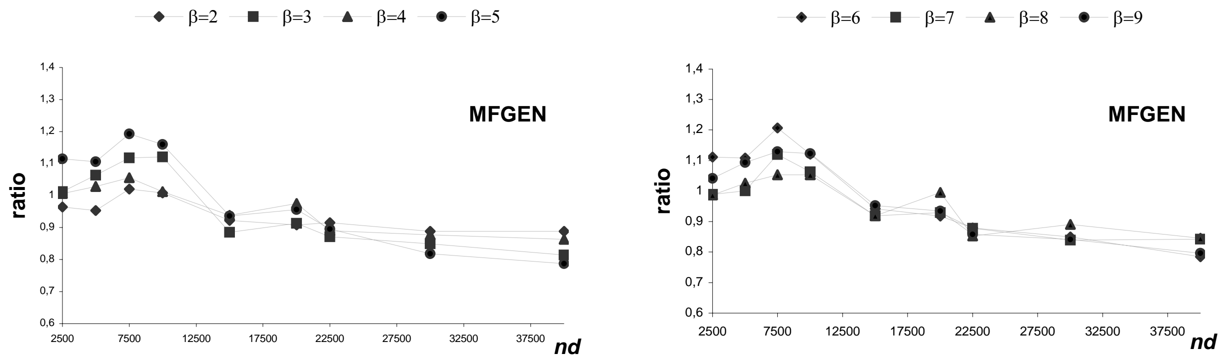


Figura 2.6. Ratio CPUV/CPU frente a nd para problemas generados con MFGEN.

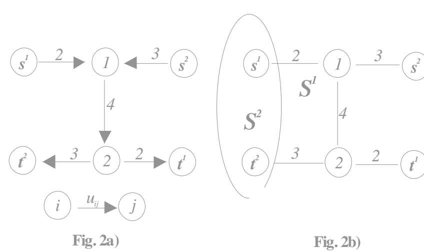


Fig. 2a) Fig. 2b)

Capítulo 3

Problemas del Biflujo Máximo

1. Introducción

Cuando se consideran diferentes tipos de bienes sobre una misma red aparecen los denominados problemas de flujos múltiples. En general, para el caso de tres o más tipos de flujos, no se verifican teoremas análogos al teorema de flujo-máximo corte-mínimo dado por Ford y Fulkerson [28]. Sin embargo, para el caso de dos bienes, Hu [43] prueba el teorema de biflujo-máximo corte-mínimo. Un resultado análogo es aportado por Seymour [72].

En este capítulo consideramos el problema de biflujo máximo en el caso no dirigido. La formulación de este problema hace uso de una red dirigida donde los valores de los flujos pueden ser negativos. Esta formulación inicial es común con las utilizadas por Hu [43], Sakarovitch [65], Rothschild y Whinston [64] y Seymour [72]. En estas condiciones desarrollamos una nueva demostración del teorema de biflujo-máximo corte-mínimo, introduciendo cambios de variables similares a los considerados por Sakarovitch [65]. Este planteamiento permite dividir el problema original en dos problemas de flujo máximo, de tal manera que la demostración alternativa del anterior teorema se base en los resultados y estructuras conocidas del ya estudiado clásico problema de flujo máximo.

Los procedimientos dados en Hu [43], Sakarovitch [65], Rothschild y Whinston [64] se basan en la aplicación reiterada del algoritmo de Ford y Fulkerson [28] sobre redes cuyos arcos tienen cotas inferiores distintas de cero e incluso negativas. Los cambios en las variables que introducimos implican que todas las cotas inferiores sean cero y que, por tanto, en el proceso de resolución se puedan utilizar las herramientas clásicas desarrolladas para flujos de un solo tipo. Todo esto redunda en un algoritmo de complejidad $O(nm \log U)$.

Un caso especial del problema de biflujo máximo, es el problema de biflujo máximo simétrico, en donde, además de enviar la mayor cantidad de flujo entre dos pares de nodos de la red, el problema precisa que la cantidad de flujo entre ambos pares ha de

coincidir. En este capítulo, presentamos un algoritmo de complejidad O($nm \log U$) basándonos en las ideas introducidas para el problema original.

Finalmente, consideramos el problema de biflujo máximo biobjetivo como una extensión natural del problema de biflujo máximo. El caso biobjetivo considera la maximización vectorial del par de flujos en lugar de la suma del mencionado par. En este capítulo está extensión es estudiada caracterizando el conjunto de soluciones eficientes extremas en el espacio objetivo. Como demostraremos, el conjunto de soluciones eficientes en el espacio de decisiones se corresponde con el conjunto todas las soluciones alternativas óptimas del problema de biflujo máximo.

2. Preliminares y formalización del problema

Sea la red dirigida *G=(V,A)* con *n* nodos y *m* arcos. Distinguimos cuatro nodos especiales en *G*: los nodos fuentes s^1, s^2 y los nodos sumideros t^1, t^2. Al igual que en Sakarovitch [65], consideraremos que la red es antisimétrica, es decir, $(i,j) \in A \Rightarrow (j,i) \notin A$. Recordaremos que, dado cualquier nodo *i*, definimos el conjunto $\text{Pred}(i) = \{j \in V / (j,i) \in A\}$ y el conjunto $\text{Suc}(i) = \{j \in V / (i,j) \in A\}$. Cada arco $(i,j) \in A$ tiene asociado una capacidad u_{ij}. Definimos la lista de arcos que salen del nodo *i* como $\text{A}(i) = \{(i,j) \in A : j \in Suc(i)\}$. Denotaremos por *U* al máximo valor del conjunto $\{u_{ij} / (i,j) \in A\}$.

Un *biflujo* es un par $x = (x^1, x^2)$, donde $x^k : A \rightarrow R$ con *k*=1,2, que satisface:

$$\sum_{j \in Suc(i)} x_{ij}^k - \sum_{j \in Pred(i)} x_{ji}^k = \begin{cases} f^k & si\ \ i = s^k \\ 0 & i \in V - \{s^k, t^k\} \\ -f^k & si\ \ i = t^k \end{cases} \qquad k = 1,2 \qquad (3.1a)$$

$$|x_{ij}^1| + |x_{ij}^2| \leq u_{ij},\ (i,j) \in A \qquad (3.1b)$$

para algún $f=(f^1,f^2)$ con $f^k \geq 0$ para k=1,2.

El problema de biflujo máximo (BFM) consiste en encontrar un biflujo x tal que la suma f^1+f^2 sea máxima. En otras palabras, el problema BFM consiste en enviar simultáneamente la mayor cantidad posible de flujo de la fuente s^1 al sumidero t^1 y de la fuente s^2 al sumidero t^2, satisfaciendo las restricciones de capacidades (3.1b) y la de conservación del flujo (3.1a).

Debemos notar que las restricciones (3.1b) permiten la posibilidad de valores negativos en el biflujo. Esto significa que si el valor del k-ésimo flujo x_{ij}^k es negativo para el arco $(i,j)\in A$, el flujo atraviesa dicho arco en sentido opuesto.

El problema de biflujo máximo simétrico (BFMS) se plantea en los mismos términos que el problema BFM, pero con la restricción añadida de que la cantidad enviada entre los nodos s^1 y t^1 debe coincidir con la que se envíe entre los nodos s^2 y t^2, es decir, se ha de cumplir que $f^1=f^2$.

Evidentemente, $s^1 \neq t^1$ y $s^2 \neq t^2$. Asumiremos sin pérdida de generalidad que $s^1 \notin \{s^2,t^2\}$ y $s^1 \notin \{s^2,t^2\}$. En cualquier otro caso, bastaría con introducir una nueva fuente o un nuevo sumidero, o ambos, de tal manera que supliesen a los que faltan, e introducir los arcos necesarios para conectarlos. En la Tabla 3.1, se indican los posibles casos, así como los nodos y arcos a introducir con sus correspondientes capacidades.

Caso	Modificación de la Red
$s^1 \neq s^2$ y $t^1 = t^2$	Añade el nodo t^{2*} conectado a t^1 mediante el arco (t^1, t^{2*}) de capacidad igual a la suma de las capacidades de los arcos que llegan a t^1.
$s^1 = s^2$ y $t^1 \neq t^2$	Añade el nodo s^{2*} conectado a s^1 mediante el arco (s^{2*}, s^1) de capacidad igual a la suma de las capacidades de los arcos que salen de s^1.
$s^1 = s^2$ y $t^1 = t^2$	En este caso, tenemos el problema clásico de Flujo Máximo entre un nodo fuente y un nodo sumidero.
$s^1 = t^2$ y $s^2 \neq t^1$	Divide el nodo s^1 en: la fuente s^{1*} con los arcos que salen de s^1 y el nodo sumidero t^{2*} con los arcos que llegan a s^1. Añade el arco (s^{1*}, t^{2*}) con capacidad infinita.
$s^1 = t^2$ y $t^1 = t^2$	Divide el nodo s^2 en: la fuente s^{2*} con los arcos que salen de s^2 y el nodo sumidero t^{1*} con los arcos que llegan a s^2. Añade el arco (s^{2*}, t^{1*}) con capacidad infinita.

$s^1 = t^2$ y $t^1 = t^2$	Resuelve este caso como la combinación de los dos anteriores.

Tabla 3.1. Modificaciones de la red para casos especiales del problema BFM

Las restricciones (3.1b) rompen la propiedad de unimodularidad exhibida en los problemas de flujos en redes con un único bien. Por lo tanto, aún bajo la asunción de que los valores de las capacidades son todos enteros, el patrón de biflujo óptimo puede ser no entero. El ejemplo de la Figura 3.1 muestra esta situación. En dicho ejemplo las capacidades de los arcos son todas iguales a 1 y el patrón de biflujo óptimo, con valor máximo de f^1+f^2=2, es el que aparece en dicha figura.

Lo que si que es cierto (se verá más adelante) es que, en las mismas condiciones, el valor óptimo de f^1+f^2 siempre es entero.

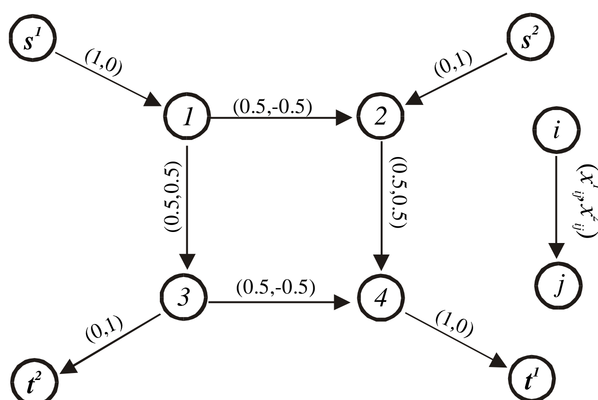


Figura 3.1. Biflujo Máximo no entero.

En el caso de un solo tipo de flujo, Ford y Fulkerson [28] demuestran que la mayor cantidad de flujo que se puede enviar desde el nodo fuente *s* al nodo sumidero *t* coincide con la menor de las capacidades de todos los *s-t* cortes de la red (Teorema del *Flujo-Máximo Corte-Mínimo*). En el caso del problema de Biflujo Máximo necesitamos ver si existe un teorema similar. Hu [43] da el siguiente resultado para el caso de un grafo no dirigido:

Teorema 3.1. (*teorema del biflujo-máximo corte-mínimo*) *El valor máximo de* (f^1+f^2) = *min* ($u\left[S^1,\overline{S}^1\right]$, $u\left[S^2,\overline{S}^2\right]$), *donde* $\left[S^1,\overline{S}^1\right]$ *recorre los posibles cortes tales que* $s^1,s^2 \in S^1$ *y* $t^1,t^2 \in \overline{S}^1$ *y* $\left[S^2,\overline{S}^2\right]$ *recorre los posibles cortes tales que* $s^1,t^2 \in S^2$ *y* $t^1,s^2 \in \overline{S}^2$.

La demostración de este teorema es probada por Hu [43] mediante un algoritmo para construir biflujos. Nosotros demostraremos este teorema de una manera alternativa a partir de una formulación equivalente del problema (3.1). Del teorema de Hu se concluye que si las capacidades son todas enteras entonces el valor máximo de f^1+f^2 es entero.

3. Formulación equivalente del problema BFM

Para obtener la formulación equivalente del problema (3.1), proponemos, de manera similar que en Sakarovitch [65], un cambio de variables con el fin de poder separarlo en dos problemas de flujo máximo con un único bien. Los cambios de variables propuestos por Sakarovitch, implican que las variables de flujos de los nuevos problemas están acotadas inferiormente por valores negativos. Esto dificulta notablemente su resolución, debido a que, previamente, se necesita resolver un problema de factibilidad. Además, los valores de las capacidades y del vector de constantes pueden ser no enteros, aún cuando las capacidades sean todas enteras. Con el cambio de variables que proponemos estas dificultades no tienen lugar ya que las cotas inferiores son todas iguales a cero y se mantiene la integridad de las constantes del problema. El referido cambio consiste en:

$$x_{ij}^1 = \frac{(z_{ij}^1 + z_{ij}^2)}{2} - u_{ij};\quad x_{ij}^2 = \frac{(z_{ij}^1 - z_{ij}^2)}{2}$$

$$z_{ij}^1 - u_{ij} = x_{ij}^1 + x_{ij}^2;\quad z_{ij}^2 - u_{ij} = x_{ij}^1 - x_{ij}^2$$

Veamos cómo se modifican las restricciones (3.1b) cuando se introduce este cambio de variables. Para ello, podemos escribir las restricciones (3.1b) de la manera siguiente:

$$\left|x_{ij}^1\right| + \left|x_{ij}^2\right| \le u_{ij} \Leftrightarrow \begin{cases} -u_{ij} \le x_{ij}^1 + x_{ij}^2 \le u_{ij} \\ -u_{ij} \le x_{ij}^1 - x_{ij}^2 \le u_{ij} \end{cases} \quad (3.1b')$$

de donde se obtiene que $0 \le z_{ij}^k \le 2u_{ij}$ con k=1,2. Realizando el cambio de variables y posteriormente sumando y restando las restricciones

de conservación de flujo, llegamos al siguiente problema equivalente de (3.1):

$$\text{maximizar} \quad f^1 + f^2$$
$$\text{sujeto a :}$$

$$\sum_{j\in Suc(i)} (z^1_{ij} - u_{ij}) - \sum_{j\in Pred(i)} (z^1_{ji} - u_{ji}) = \begin{cases} f^1 & \text{si } i = s^1 \\ f^2 & \text{si } i = s^2 \\ 0 & \forall \text{i} \neq \{\text{s}^1, s^2, t^1, t^2\} \\ -f^1 & \text{si } i = t^1 \\ -f^2 & \text{si } i = t^2 \end{cases} \quad (3.2a)$$

$$\sum_{j\in Suc(i)} (z^2_{ij} - u_{ij}) - \sum_{j\in Pred(i)} (z^2_{ji} - u_{ji}) = \begin{cases} f^1 & \text{si } i = s^1 \\ -f^2 & \text{si } i = s^2 \\ 0 & \forall \text{i} \neq \{\text{s}^1, s^2, t^1, t^2\} \\ -f^1 & \text{si } i = t^1 \\ f^2 & \text{si } i = t^2 \end{cases} \quad (3.2b)$$

$$0 \leq z^k_{ij} \leq 2u_{ij}, \; k = 1,2 \;\; \forall (i, j) \in A \qquad (3.2c)$$

P2

En el problema *P2* se puede observar que las restricciones pueden ser separadas en aquellas que únicamente afectan a cada una de las variables z^k con k=1,2. Sin embargo, el valor de la función objetivo no es separable. Separaremos estos dos problemas, llamando *P2a* al problema con las restricciones que únicamente contienen las variables z^1 y *P2b* al relacionado con las variables z^2. Para ello denotamos, para todo $i \in V$, $\delta_i = \sum_{j\in Suc(i)} u_{ij} - \sum_{j\in Pred(i)} u_{ij}$, y escribimos ambos problemas de la manera siguiente:

$$\text{maximizar} \quad f^1 + f^2$$
$$\text{sujeto a :}$$

$$\sum_{j\in Suc(i)} z^1_{ij} - \sum_{j\in Pred(i)} z^1_{ji} = \begin{cases} f^1 + \delta_{s^1} & \text{si } i = s^1 \\ f^2 + \delta_{s^2} & \text{si } i = s^2 \\ \delta_\text{i} & \forall \text{i} \in \text{V} \neq \{\text{s}^1, s^2, t^1, t^2\} \\ -f^1 + \delta_{t^1} & \text{si } i = t^1 \\ -f^2 + \delta_{t^2} & \text{si } i = t^2 \end{cases}$$

$$0 \leq z^1_{ij} \leq 2u_{ij}, \;\; \forall (i, j) \in A$$

P2a

$$
\begin{aligned}
&\text{maximizar} \quad f^1 + f^2 \\
&\text{sujeto a :} \\
&\sum_{j \in Suc(i)} z_{ij}^2 - \sum_{j \in Pred(i)} z_{ji}^2 = \begin{cases} f^1 + \delta_{s^1} & \text{si } i = s^1 \\ -f^2 + \delta_{s^2} & \text{si } i = s^2 \\ \delta_i & \forall i \in V \neq \{s^1, s^2, t^1, t^2\} \\ -f^1 + \delta_{t^1} & \text{si } i = t^1 \\ f^2 + \delta_{t^2} & \text{si } i = t^2 \end{cases} \qquad \boldsymbol{P2b} \\
&0 \le z_{ij}^2 \le 2u_{ij}, \quad \forall (i,j) \in A
\end{aligned}
$$

El problema *P2a* se refiere al problema de flujo máximo de un único bien entre los nodos fuentes $\{s^1, s^2\}$ y los nodos sumideros $\{t^1, t^2\}$, donde, además, se precisa que una cantidad de flujo igual a $|\delta_i|$ salga de los nodos (si $\delta_i > 0$) o llegue a los nodos (si $\delta_i < 0$) (subproblema de factibilidad). De manera similar, el problema *P2b* hace referencia al problema de flujo máximo de un único bien entre los nodos fuentes $\{s^1, t^2\}$ y los nodos sumideros $\{t^1, s^2\}$, además del subproblema de factibilidad anteriormente mencionado. Evidentemente, si la solución óptima de estos dos problemas es la misma, hemos solucionado el problema *P2*.

Lema 3.1. *El valor óptimo de P2 coincide con el mínimo de los valores óptimos de P2a y P2b.*

Demostración. La única diferencia entre los problemas *P2a* y *P2b* estriba en las restricciones que afectan a los nodos s^2 y t^2. Además, existe al menos una solución de ambos problemas para la que el valor de f^1 es máximo, dado que ambos problemas se refieren a la misma red. Sean entonces f_a^2 y f_b^2 los valores máximos alcanzados para cada uno de los problemas cuando se ha alcanzado el valor f^1 en ambos. Si $f_a^2 = f_b^2$, se satisfacen las restricciones del problema *P2* y, por tanto, el valor óptimo de *P2* es $f^1 + f_a^2 = f^1 + f_b^2$. Si $f_a^2 > f_b^2$, también las restricciones de *P2* se satisfacen para $f^2 = f_b^2$ y, si $f_a^2 < f_b^2$, se satisfacen para $f^2 = f_a^2$. En estos casos, $f^1 + f^2 = f^1 + \min\{f_a^2, f_b^2\}$.□

El siguiente resultado establece la relación entre el valor óptimo del problema *P2a* con el problema de flujo máximo entre los nodos s^1, s^2 y los nodos t^1, t^2 (en el grafo de partida y con las capacidades originales).

Lema 3.2. *El valor óptimo de P2a no puede exceder de la capacidad de cualquier corte* $[S,\bar{S}]$ *de la red original, donde la capacidad viene dada por* $u^*[S,\bar{S}] = \sum_{(i,j)\in(S,\bar{S})} u_{ij} + \sum_{(i,j)\in(\bar{S},S)} u_{ij}$.

Demostración. Sea $[S,\bar{S}]$ un corte cualquiera del grafo asociado al problema *P2a* tal que $s^1, s^2 \in S$ y $t^1, t^2 \in \bar{S}$. Consideremos la suma de las ecuaciones de conservación para los nodos del conjunto *S*, tenemos:

$$f^1 + f^2 + \sum_{i\in S} \delta_i = \sum_{i\in S} (\sum_{j\in Suc(i)} z^1_{ij} - \sum_{j\in Pred(i)} z^1_{ji})$$

$$f^1 + f^2 + \sum_{i\in S} (\sum_{j\in Suc(i)} u_{ij} - \sum_{j\in Pred(i)} u_{ji}) = \sum_{i\in S} (\sum_{j\in Suc(i)} z^1_{ij} - \sum_{j\in Pred(i)} z^1_{ji})$$

Debemos notar que, en las sumas anteriores, únicamente intervienen los arcos $(i,j)\in(S,\bar{S})$ y los arcos $(i,j)\in(\bar{S},S)$. De esta manera tenemos que:

$$f^1 + f^2 + \sum_{(i,j)\in(S,\bar{S})} u_{ij} - \sum_{(j,i)\in(\bar{S},S)} u_{ji} = \sum_{(i,j)\in(S,\bar{S})} z^1_{ij} - \sum_{(j,i)\in(\bar{S},S)} z^1_{ji}$$

Debido a que $0 \le z^1_{ij} \le 2u_{ij}$ y a que interesa maximizar:

$$f^1 + f^2 + \sum_{(i,j)\in(S,\bar{S})} u_{ij} - \sum_{(j,i)\in(\bar{S},S)} u_{ji} \le \sum_{(i,j)\in(S,\bar{S})} 2u_{ij}$$

de donde resulta $f^1 + f^2 \le \sum_{(i,j)\in(S,\bar{S})} u_{ij} + \sum_{(j,i)\in(\bar{S},S)} u_{ji}$.

Si definimos la capacidad de un corte como $u^*[S,\bar{S}] = \sum_{(i,j)\in(S,\bar{S})} u_{ij} + \sum_{(i,j)\in(\bar{S},S)} u_{ij}$, entonces $f^1 + f^2 \le u^*[S,\bar{S}]$.□

Teorema 3.2. *El valor óptimo de P2a coincide con la menor de las capacidades de los cortes del grafo no dirigido obtenido cuando no se consideran direcciones en el grafo original.*

Demostración. Sea el grafo original *G*=(*V*,*A*), y sea el grafo *G´*=(*V*,*A´*) obtenido a partir de *G* cuando no se consideran las direcciones de los arcos. Consideremos el problema de flujo máximo con un único bien entre los nodos fuentes s^1, s^2 y los nodos sumideros t^1, t^2 en *G´*. Por el teorema de flujo-máximo corte-mínimo, existe un corte $[S,\overline{S}]$ con $s^1,s^2 \in S$ y $t^1,t^2 \in \overline{S}$ tal que $f^1+f^2 = u[S,\overline{S}] = \sum_{(i,j)\in(S,\overline{S})} u_{ij}$. Si de nuevo consideramos las direcciones, algunos de los arcos en el corte $[S,\overline{S}]$ en la red original *G* son de la forma $(S,\overline{S})$ y el resto son de la forma $(\overline{S},S)$. Por lo tanto, podemos escribir $u[S,\overline{S}] = \sum_{(i,j)\in(S,\overline{S})} u_{ij} + \sum_{(i,j)\in(\overline{S},S)} u_{ij}$. Consecuentemente $u[S,\overline{S}] = u^*[S,\overline{S}]$ y por lo tanto, $f^1+f^2 = u^*[S,\overline{S}]$.□

De igual manera, se puede demostrar que la solución óptima del problema *P2b* coincide con la solución óptima del problema de flujo máximo con un único bien entre los nodos fuentes s^1, t^2 y los nodos sumideros t^1, s^2 cuando no se consideran direcciones en los arcos. De los anteriores resultados podemos escribir el teorema de Hu de la siguiente manera:

Teorema 3.3: *(biflujo-máximo corte-mínimo) El biflujo máximo en un Grafo G=(V,A) coincide con el mínimo de las capacidades de los cortes de la forma* $[S^1,\overline{S}^1]$ *y* $[S^2,\overline{S}^2]$ *en el grafo no dirigido G´=(V,A´), donde* $s^1,s^2 \in S^1$ *y* $s^1,t^2 \in S^2$*, es decir,* $f^1+f^2 = min(u[S^1,\overline{S}^1], u[S^2,\overline{S}^2])$ *en G´=(V,A´).*

Demostración: La demostración es inmediata a partir del lema 3.1 y del teorema 3.2. □

En el ejemplo dado en la Figura 3.2, puede comprobarse el enunciado del teorema. La capacidad del corte $u\left[S^1,\overline{S}^1\right]$ = 4 y la capacidad del corte $u\left[S^2,\overline{S}^2\right]$ = 5, de donde f^1+f^2= min (4,5) = 4.

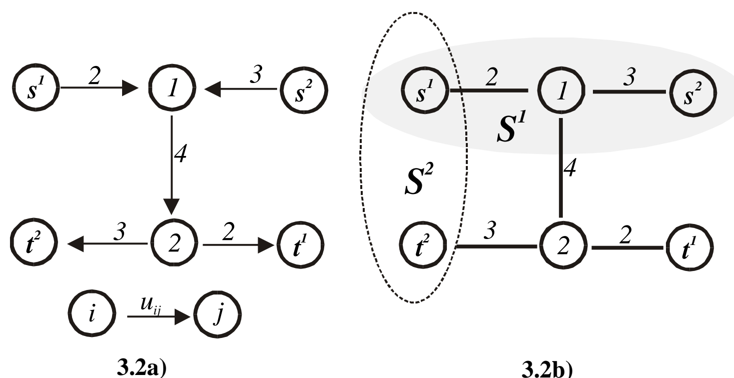


Figura 3.2.

3.1 Cambio de variables alternativo

El Teorema 3.3, sugiere haber considerado el siguiente cambio de variables:

$$x^1_{ij} + x^2_{ij} = z^1_{ij} - z^1_{ji}\,;\;\; x^1_{ij} - x^2_{ij} = z^2_{ij} - z^2_{ji}\,,$$

$$x^1_{ij} = (z^1_{ij} - z^1_{ji} + z^2_{ij} - z^2_{ji})/2\,;\;\; x^2_{ij} = (z^1_{ij} - z^1_{ji} - z^2_{ij} + z^2_{ji})/2$$

Si consideramos las restricciones (3.1b´) tenemos que las restricciones asociadas a las variables *z* son $-u_{ij} \le z^k_{ij} - z^k_{ji} \le u_{ij}$ con k=1,2.

Estas restricciones podemos escribirlas como $0 \le z^k_{ij} \le u_{ij}$ y $0 \le z^k_{ji} \le u_{ij}$, con k=1,2. Es decir, por cada arco de la red antisimétrica original, introducimos los arcos (i,j) y (j,i) en las respectivas redes asociadas con las variables z^k. Para cada una de estas redes, denotaremos los nodos adyacentes o vecinos de un nodo *i* por $N(i) = \{Suc(i) \cup Pred(i)\}$. Si llevamos a cabo este cambio de variables, obtendríamos los siguientes problemas equivalentes a los problemas *P2a* y *P2b*:

$$
\begin{aligned}
&\text{maximizar} \quad f^1 + f^2 \\
&\text{sujeto a :} \\
&\sum_{j \in N(i)} z_{ij}^1 - \sum_{j \in N(i)} z_{ji}^1 = \begin{cases} f^1 & \text{si } i = s^1 \\ f^2 & \text{si } i = s^2 \\ 0 & \forall \mathrm{i} \in \mathrm{V} \neq \{\mathrm{s}^1, s^2, t^1, t^2\} \\ -f^1 & \text{si } i = t^1 \\ -f^2 & \text{si } i = t^2 \end{cases} \quad \mathbf{P2a'} \\
&0 \leq z_{ij}^1 \leq u_{ij}, \quad \forall (i,j) \in A \\
&0 \leq z_{ji}^1 \leq u_{ij}, \quad \forall (i,j) \in A
\end{aligned}
$$

$$
\begin{aligned}
&\text{maximizar} \quad f^1 + f^2 \\
&\text{sujeto a :} \\
&\sum_{j \in N(i)} z_{ij}^2 - \sum_{j \in N(i)} z_{ji}^2 = \begin{cases} f^1 & \text{si } i = s^1 \\ -f^2 & \text{si } i = s^2 \\ 0 & \forall \mathrm{i} \in \mathrm{V} \neq \{\mathrm{s}^1, s^2, t^1, t^2\} \\ -f^1 & \text{si } i = t^1 \\ f^2 & \text{si } i = t^2 \end{cases} \quad \mathbf{P2b'} \\
&0 \leq z_{ij}^2 \leq u_{ij}, \quad \forall (i,j) \in A \\
&0 \leq z_{ji}^2 \leq u_{ij}, \quad \forall (i,j) \in A
\end{aligned}
$$

El problema *P2a´* se refiere al problema de flujo máximo de un único bien entre las fuentes s^1, s^2 y los sumideros t^1, t^2 en la red de partida sin considerar direcciones en los arcos. De manera similar, el problema *P2b'* hace referencia al problema de flujo máximo de un único bien entre las fuentes s^1, t^2 a los sumideros t^1, s^2 en la correspondiente red no dirigida.

La resolución del problema de biflujo máximo la efectuaremos, sin embargo, utilizando los problemas *P2a* y *P2b*. Esto implica que podremos desarrollar un algoritmo teniendo presente que las cotas inferiores de los arcos son todas iguales a cero y que los valores de los flujos correspondientes a cada uno de esos problemas son enteros siempre que las capacidades lo sean. Esto simplifica la resolución del problema si lo comparamos con los algoritmos introducidos por Hu[43] y por Sakarovitch[65].

Para la resolución del problema *P2a* y *P2b*, añadimos dos nuevos vértices *h* y *h'* en el grafo original. Sean $I^{+}=\{i\in V:\delta_i>0\}$ y $I^{-}=\{i\in V:\delta_i<0\}$. Entonces añadimos al grafo los arcos (h,i) con $u_{hi}=\delta_i$ para todo $i\in I^{+}$ y los arcos (i,h') con $u_{ih'}=-\delta_i$ para todo $i\in I^{+}$. Sean $\delta_h=\sum_{i\in I^{+}}\delta^i$ y $\delta_{h'}=\sum_{i\in I^{-}}\delta^i$ (evidentemente, $\delta_h=-\delta_{h'}$). Introduciendo estos dos nuevos nodos en la formulación de ambos problemas, obtenemos:

$$\begin{aligned}
&\text{maximizar} \quad f^1+f^2\\
&\text{sujeto a}:\\
&\sum_{j\in Suc(i)} z_{ij}^1-\sum_{j\in Pred(i)} z_{ij}^1=\begin{cases} f^1 & \text{si } i=s^1\\ f^2 & \text{si } i=s^2\\ \delta_h & \text{si } i=h\\ 0 & \forall i\in V\neq\{s^1,s^2,t^1,t^2\}\\ -\delta_h & \text{si } i=h'\\ -f^1 & \text{si } i=t^1\\ -f^2 & \text{si } i=t^2\end{cases} \qquad \textbf{P3a}\\
&0\le z_{ij}^1\le 2u_{ij},\quad \forall(i,j)\in A\\
&0\le z_{hj}^1\le u_{hj},\ \forall j\in I^{+}\ \text{y}\ 0\le z_{jh'}^1\le u_{jh'},\quad \forall j\in I^{-}
\end{aligned}$$

$$\begin{aligned}
&\text{maximizar} \quad f^1+f^2\\
&\text{sujeto a}:\\
&\sum_{j\in Suc(i)} z_{ij}^2-\sum_{j\in Pred(i)} z_{ij}^2=\begin{cases} f^1 & \text{si } i=s^1\\ -f^2 & \text{si } i=s^2\\ \delta_h & \text{si } i=h\\ 0 & \forall i\in V\neq\{s^1,s^2,t^1,t^2\}\\ -\delta_h & \text{si } i=h'\\ -f^1 & \text{si } i=t^1\\ f^2 & \text{si } i=t^2\end{cases} \qquad \textbf{P3b}\\
&0\le z_{ij}^2\le 2u_{ij},\quad \forall(i,j)\in A\\
&0\le z_{hj}^2\le u_{hj},\ \forall j\in I^{+}\ \text{y}\ 0\le z_{jh'}^1\le u_{jh'},\quad \forall j\in I^{-}
\end{aligned}$$

La inclusión de los nodos *h* y *h'* supone la consideración de un problema de factibilidad entre ambos vértices; es decir, han de ser enviadas δ_h unidades del nodo *h* al nodo *h'*. Es evidente que si resolvemos el problema de flujo máximo entre *h* y *h'*, la máxima

cantidad de flujo que se puede enviar entre estos dos nodos es justamente δ_h. Es igual de evidente que siempre es posible enviar esas δ_h unidades de flujo de *h* a *h'*.

4. Algoritmo para obtener un biflujo máximo

Si observamos el problema *P3a* y *P3b*, ambos problemas requieren enviar la mayor cantidad de flujo posible de los nodos fuentes $\{s^1, h\}$ a los nodos sumideros $\{t^1, h'\}$. Esto indica que el primer paso del algoritmo sea, justamente, obtener el flujo máximo entre estos pares de nodos. Es evidente que, para este primer paso, se podrá utilizar cualquier algoritmo de flujo máximo. Una vez resuelto este primer paso, consideramos dos redes residuales, cada una de ellas asociadas a las variables z^k con k=1,2. Sean R^1 y R^2 las correspondientes redes residuales una vez se han enviado las $f^1 + \delta_h$ unidades de flujo mencionadas con anterioridad. Para obtener la solución del problema de biflujo máximo, necesitamos enviar la misma cantidad de flujo f^2 de s^2 a t^2 en R^1 y de t^2 a s^2 en R^2. Esto implicará identificar caminos incrementales en ambas redes residuales y enviar el mínimo de las capacidades residuales a través de los correspondientes caminos en ambas redes. Esta operación, implica una actualización de las capacidades residuales de los arcos de ambos caminos.

Para identificar los respectivos caminos, proponemos el uso de dos etiquetas distancias como la introducida por Goldberg [38]. Denotaremos por $d^1(i)$ a la etiqueta distancia del nodo *i* en R^1 ($d^1(t = t^2) = 0$) y por $d^2(i)$ a la etiqueta distancia del mismo nodo en R^2 ($d^2(t = s^2) = 0$).

Con el fin de no tener que calcular las capacidades residuales de los caminos incrementales y, posteriormente, actualizar las capacidades residuales de sus arcos, proponemos introducir una escala en las capacidades. Para definirla, usamos el parámetro de escala Δ. Por tanto, cada red residual $R^k(\Delta)$, con k=1,2, únicamente contiene arcos cuya capacidad residual es, al menos, Δ. De esta

manera, todos los caminos incrementales en dichas redes tienen una capacidad residual de al menos *Δ*. Como hemos introducido una escala, diremos que un arco (i,j) es *Δ-admisible* en $R^k(\Delta)$ si $d(i)=d(j)+1$ y $r_{ij}^k \geq \Delta$. Por lo tanto, en una fase *Δ* se identificarán caminos incrementales *Δ-admisibles* en R^1 y en R^2 de manera simultánea.

Definición 3.1. Un biflujo es *Δ-óptimo*, si no existe ningún camino incremental con capacidad residual de al menos *Δ* en R^1 y en R^2 simultáneamente.

El algoritmo que proponemos trabaja en fases de escalado. Cada fase tiene un *Δ* fijo y, cuando finaliza una fase, comienza la siguiente con *Δ=Δ/2*. Al principio $\Delta = 2^{\lceil \log 2U \rceil}$, es decir, es igual al menor entero potencia de dos que es mayor o igual que $2U$. La última fase del algoritmo ocurre cuando *Δ=1*. En esta última fase $R^k(\Delta)=R^K$, con k=1,2, y, por tanto, al finalizar el algoritmo se determina el biflujo máximo. De esta manera, el algoritmo realiza $\lceil \log U \rceil + 1$ fases de escalado. Los procedimientos necesarios para el algoritmo así como un esquema del mismo son dados a continuación:

```
Procedure Avanzar(i,d,R,Δ,pred);
  begin
  Sea (i,j) un arco Δ-admisible;
  r_ij = r_ij − Δ; r_ji = r_ji + Δ; pred(j) := i; i := j
  end;

Procedure Restablecer(i,s,R,Δ,pred);
  begin
  while (i≠s) do
    begin
    j := i; i := pred(i);
    r_ij = r_ij + Δ;r_ji = r_ji − Δ
    end
  end;

Procedure Retroceder(i,d,R,Δ,pred,s);
  begin
  d(i) := min{d(j) + 1 : (i, j) ∈ A'(i) y r_ij ≥ Δ};
  if i ≠ s then
    begin
    j := i; i := pred(i);
    r_ij = r_ij + Δ; r_ji = r_ji − Δ
```

```
    end
  end;
Algoritmo Biflujo Máximo(BFM);
  begin
  Sea G=(V,A); Consideremos el grafo G'(V',A') correspondiente
  para el problema P3a y P3b; Sean las capacidades u' las
  definidas en dicha red;
  Sea Z^1 = Z^2:=0;
  Obtener mediante la aplicación de cualquier algoritmo de flujo
  máximo, la mayor cantidad de flujo que se puede enviar desde
  las fuentes s^1 y h a los sumideros t^1 y h'; Sea Z^1 el
  correspondiente flujo; Sea f^1 la cantidad de flujo enviada de
  s^1 a t^1; f^2:=0;
  Hacer Z^2 := Z^1; r^1_ij = r^2_ij := u'_ij - z^1_ij + z^1_ji  ∀(i,j) ∈ A';
  Δ := 2^⌈log 2U⌉;
  while Δ ≥ 1 do
    begin
    d^1(t^2) := 0; Obtener d^1 mediante un BFS inverso en R^1
    empezando en t^2;
    d^2(s^2) := 0; Obtener d^2 mediante un BFS inverso en R^2
    empezando en s^2;
    i^1:= s^2; i^2:= t^2;
    while (d^1(s^2) < n) and (d^2(t^2) < n) do
      begin
      if (i^1 ≠ t^2) then
        if (i^1 tiene un arco Δ-admisible) then
          Avanzar(i^1,d^1,R^1,Δ,pred^1)
        else Retroceder(i^1,d^1,R^1,Δ,pred^1,s^2);
      if (i^2 ≠ s^2) then
        if (i^2 tiene un arco Δ-admisible) then
          Avanzar(i^2,d^2,R^2,Δ,pred^2)
        else Retroceder(i^2,d^2,R^2,Δ,pred^2,t^2);
      if (i^1 = t^2) and (i^2=s^2) then i^1:= s^2; i^2:= t^2;f^2:=f^2+Δ
      end;
    if (i^1 ≠ s^2) then Restablecer(i^1,s^2,R^1,Δ,pred^1);
    if (i^2 ≠ t^2) then Restablecer(i^2,t^2,R^2,Δ,pred^2);
    Δ := Δ / 2
    end
  end.
```

Una vez obtenido el flujo máximo entre los nodos fuentes s^1, h y los nodos sumideros t^1, h', comienzan las fases de escalado. Cada fase de escalado procede enviando Δ unidades de flujo a lo largo de caminos Δ-admisibles. Para ello debe identificar simultáneamente un camino Δ-admisible en cada red residual $R^k(\Delta)$ con k=1,2. Cada

fase de escalado comienza calculando las etiquetas distancias exactas d^k con k=1,2. El algoritmo mantiene un índice de predecesor para los nodos de cada red $R^k(\Delta)$, de tal manera que $pred^k(i^k)$ almacena el nodo anterior al nodo i^k en el camino Δ-admisible actual de $R^k(\Delta)$. El algoritmo realiza, iterativamente, pasos *Avanzar* o *Retroceder* sobre el último nodo de cada camino admisible parcial. Si del nodo actual i^k sale un arco admisible (i^k, j^k), entonces realiza un paso *Avanzar* y añade este arco al camino parcial actual. Cada paso *Avanzar* actualiza las capacidades residuales del arco Δ-admisible (i^k, j^k) por $r_{ij} = r_{ij} - \Delta$ y $r_{ji} = r_{ji} + \Delta$, es decir, se envían Δ unidades de flujo a través de dicho arco. Si no se identifica un arco Δ-admisible, se realiza un paso *Retroceder* que incrementa la etiqueta distancia del nodo i^k, haciendo que el arco $(pred^k(i^k), i^k)$ sea no admisible y, por lo tanto, retrocediendo un arco sobre el camino parcial. Además, se deshace el envío anterior de Δ unidades de flujo. Así, si (i^k, j^k) está en un camino Δ-admisible, ya hemos enviado las Δ unidades de flujo y, si retrocedemos sobre él, deshacemos ese envío. Si se alcanzan los respectivos nodos sumideros, el proceso se repite comenzando en las respectivas fuentes. Es claro que esto únicamente sucede si se han detectado, a la vez, un camino Δ-admisible en R^1 y otro en R^2, incrementándose f^2 en Δ unidades de flujo. Una fase de escalado acaba cuando la etiqueta distancia de al menos un nodo fuente sea mayor o igual que n. Si esto ocurre, a lo sumo un i^k puede ser distinto de su nodo fuente, lo que significa que se han enviado Δ unidades de flujo a lo largo del camino parcial identificado para la correspondiente red residual. El procedimiento *Restablecer* se encarga de devolver esas Δ unidades de flujo. Finalmente, el algoritmo acaba cuando Δ es menor que 1. Para obtener el patrón de biflujo máximo se deshace el cambio de variables, es decir:

$$\forall (i,j) \in A, \quad \begin{array}{l} z_{ij}^1 = \max(0, u'_{ij} - r_{ij}^1) \\ z_{ij}^2 = \max(0, u'_{ij} - r_{ij}^2) \end{array} \Rightarrow x_{ij}^1 = \frac{(z_{ij}^1 + z_{ij}^2)}{2} - u_{ij} \; x_{ij}^2 = \frac{(z_{ij}^1 - z_{ij}^2)}{2}$$

4.1 Un ejemplo

A continuación aplicaremos el algoritmo anterior sobre el ejemplo introducido en la Figura 3.2. El proceso es mostrado en la Figura 3.3. La Figura 3.3a, representa la red *G'* con las capacidades *u'*. La Figura 3.3b, muestra la red residual R^1 que coincide con R^2, cuando se han enviado $f^1 + \delta_h$=8 unidades de flujo desde los nodos s^1, *h* a los nodos t^1, *h'*. Se puede observar en esta figura que únicamente hay un camino de s^2 a t^2 en R^1 de capacidad residual 2 y de t^2 a s^2 en R^2 de capacidad residual 3. El algoritmo identificará ambos caminos cuando se encuentre en la fase Δ=2 (Figura 3.3c). Tras enviar estas Δ unidades de flujo a través de estos caminos en sus respectivas redes incrementales, no queda camino incremental en $R^1(2)$ y en $R^2(2)$. Por tanto, el algoritmo pasa a la fase Δ=1 (las Figuras 3.3d y 3.3f, muestran $R^1(1)$ y $R^2(1)$). Se puede observar que en $R^1(1)$ no hay camino incremental (aunque si lo hay en $R^2(1)$) y, por lo tanto, se ha obtenido un biflujo máximo. El patrón de biflujo máximo es mostrado en la Figura 3.3f.

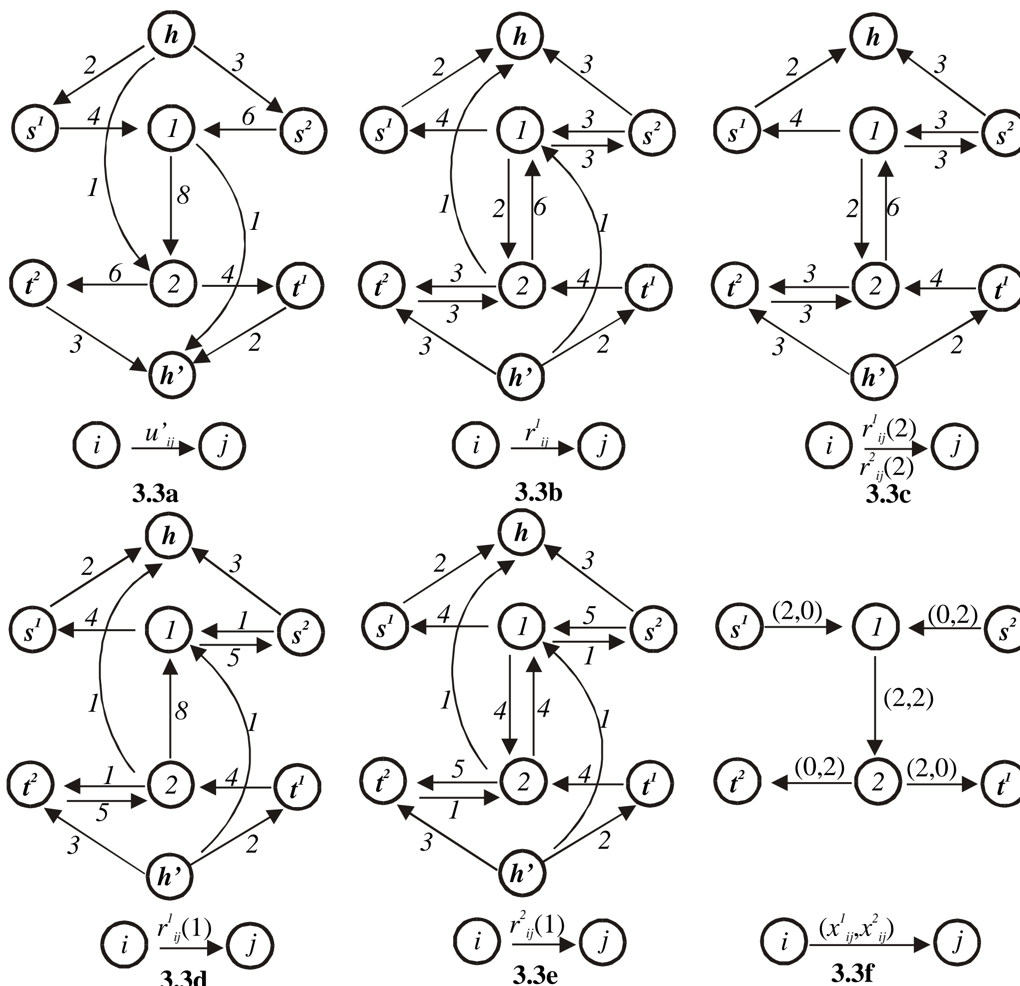


Figura 3.3. Traza del algoritmo sobre el ejemplo

4.2 Complejidad del algoritmo

En esta sección demostraremos que la complejidad del algoritmo, en el caso peor, es O($nmlog2U$). La primera operación que realiza el algoritmo es la de obtener el flujo máximo entre los nodos fuentes s^1, h y los nodos sumideros t^1, h'. Esta operación es realizada por cualquier algoritmo de flujo máximo. Por lo tanto basta con elegir uno con una complejidad teórica inferior o similar a la necesaria en las siguientes operaciones. El algoritmo presentado realiza $\lceil \log 2U \rceil$ fases de escalado. Cada fase de escalado Δ es llevada a cabo hasta que alguna de las etiquetas distancia $d^1(s^2)$ en R^1 ó $d^2(t^2)$ en R^2 es mayor o igual que n.

Lema 3.3. *Cada fase de escalado requiere un esfuerzo computacional de* O(nm).

Demostración. Supongamos que nos fijamos en la etiqueta distancia $d^1(s^2)$ en R^1 y que esta es la que determina la condición de parada de cada fase de escalado cuando $d^1(s^2) \geq n$. De esta manera, el algoritmo actualiza la etiqueta distancia $d^1(i)$ en R^1 de un nodo i a lo sumo n veces. Esto es así ya que, si un nodo tiene una distancia mayor o igual que n, este nodo no es alcanzable mediante un camino elemental desde la fuente. Además, en el peor de los casos, la etiqueta distancia de un nodo puede incrementarse cada vez en una unidad. Por tanto, el número de operaciones *Retroceder* está acotado por O(n^2), lo que implica que el esfuerzo computacional en realizar dichas operaciones es O(mn).

Veamos ahora cuál es el número de envíos de flujo. Para ello diremos que un envío es *Δ-saturante* a través de (i, j) si, después de realizarlo, la capacidad residual del arco (i, j) es estrictamente menor que Δ. Un arco puede ser Δ-saturado a lo sumo $n/2$ veces (lema 1.1). Así, sumando para todos los arcos se tiene que el número de envíos Δ-saturantes es O(nm). Esto implica que el número de envíos de flujo es a lo sumo O(nm). Como el esfuerzo computacional de cada envío de flujo es O(1), el esfuerzo total vuelve a ser O(nm). El número de llamadas a *Avanzar* está acotado por el número de envíos de flujo más el número de operaciones *Retroceder*, es decir, O(nm + n^2). Finalmente, la operación *Restablecer* es llamada a lo sumo una vez, y está requiere un esfuerzo de O(n). Por lo tanto, la complejidad de una fase, considerando que la etiqueta distancia $d^1(s^2)$ en R^1 determina la condición de parada, es O(nm).

La misma complejidad en el caso peor se obtendría si se considerara que es la etiqueta distancia $d^2(t^2)$ en R^2 la que determina la condición de parada. Como en cada fase alguna de las dos condiciones debe ocurrir, la complejidad de cada fase coincide con el máximo de las complejidades de ambas. Por lo tanto, la complejidad del algoritmo es O(nm).□

Teorema 3.4. *El algoritmo requiere un tiempo* O($nm \log U$).

Demostración. Por el lema 3.3, cada fase de escalado emplea un tiempo de O(nm). Como el número de fases de escalado es $\lceil \log 2U \rceil$, obtenemos un algoritmo de complejidad O($nm \log U$). □

Los resultados computacionales obtenidos mediante la realización de un experimento sobre el algoritmo serán mostrados más tarde. A continuación, estudiaremos el problema de biflujo máximo simétrico.

5. El problema de biflujo máximo simétrico

Llamamos problema de biflujo máximo simétrico, al problema de biflujo máximo donde se exige que $f^1 = f^2$, es decir, el problema tiene la siguiente formalización:

$$\max \; imizar \quad f^1 + f^2$$

$$\sum_{j \in Suc(i)} x_{ij}^k - \sum_{j \in Pred(i)} x_{ji}^k = \begin{cases} f^k & si \;\; i = s^k \\ 0 & i \in V - \{s^k, t^k\} \\ -f^k & si \;\; i = t^k \end{cases} \qquad k = 1{,}2 \qquad (3.3a)$$

$$\left|x_{ij}^1\right| + \left|x_{ij}^2\right| \le u_{ij},\; (i,j) \in A \qquad (3.3b)$$

$$f^1 = f^2 \qquad (3.3c)$$

El problema de biflujo máximo simétrico es un caso particular del problema de biflujo máximo y, por lo tanto, podemos resolverlo de la misma manera. Supongamos que utilizamos el algoritmo para obtener el biflujo máximo dado con anterioridad, de tal manera que entre los nodos s^1 y t^1 se ha enviado la mayor cantidad de flujo posible que denotamos por f^{1*}. Sea entonces $f^{2'}$ la mayor cantidad de flujo que se puede enviar de s^2 a t^2 cuando se envían f^{1*} unidades de flujo entre los nodos s^1 y t^1. Evidentemente, $(f^{1*}, f^{2'})$ se corresponde con una solución óptima del problema de biflujo máximo.

Lema 3.4. *Sea* (f^1, f^2) *el biflujo máximo simétrico, entonces se tiene que* $f^1 + f^2 \le f^{1*} + f^{2'}$.

Demostración. Es evidente por el teorema de biflujo-máximo corte-mínimo, que cualquier biflujo (f^1, f^2) tal que $f^1 = f^2$ debe satisfacer $f^1 + f^2 \le f^{1*} + f^{2'}$.□

La cuestión es ver si a partir del biflujo máximo $(f^{1*}, f^{2'})$ podemos obtener un biflujo máximo simétrico. La respuesta la obtenemos a partir del siguiente resultado:

Teorema 3.5. *Dado el biflujo máximo* $(f^{1*}, f^{2'})$ *se puede obtener un biflujo máximo simétrico* (f^1, f^2).

Demostración. Los posibles casos que se pueden dar son:

(i) Si $f^{1*} = f^{2'}$ (caso trivial), hemos encontrado el biflujo máximo simétrico con $f^k = f^{1*}$ para k=1,2. En este caso se tiene que $f^1 + f^2 = f^{1*} + f^{2'}$. (f^1, f^2) es máximo por el teorema 3.3.

(ii) Si $f^{1*} < f^{2'}$, tenemos que enviar – $(f^{2'} - f^{1*})$ unidades de flujo de s^2 a t^2. De esta manera $f^k = f^{1*}$ con k=1,2. En este caso, se tiene que $f^1 + f^2 = 2f^{1*} < f^{1*} + f^{2'}$; es decir, (f^1, f^2) es máximo, pues $f^1 = f^{1*}$ es el máximo valor de flujo que se puede enviar de s^1 a t^1.

(iii) Si $f^{1*} > f^{2'}$, tenemos que enviar – $(f^{1*} - f^{2'})/2$ unidades de flujo de s^1 a t^1. Por el teorema de biflujo-máximo corte-mínimo, a lo sumo se pueden enviar $(f^{1*} - f^{2'})/2$ unidades de flujo de s^2 a t^2. En este punto tenemos dos posibles subcasos:

a) Se pueden enviar las $(f^{1*} - f^{2'})/2$ de s^2 a t^2. Con lo que se obtiene un biflujo máximo simétrico tal que $f^k = (f^{1*} + f^{2'})/2$ con k=1,2. En este caso se tiene que $f^1 + f^2 = f^{1*} + f^{2'}$. (f^1, f^2) es máximo por el teorema 3.

b) Se pueden enviar ϕ unidades de s^2 a t^2 con $\phi <$ $(f^{1*} - f^{2'})/2$. En este caso, con el fin de obtener un biflujo máximo simétrico se han de enviar $-((f^{1*} - f^{2'})/2 - \phi)$ de s^1 a t^1. De esta forma se

obtiene un biflujo máximo simétrico tal que $f^k = f^{2'} + \phi$ con k=1,2. En esta ocasión se tiene que $f^1 + f^2 = 2f^{2'} + 2\phi < f^{1*} + f^{2'}$. Se obtiene entonces que (f^1, f^2) es máximo, pues $f^2 = f^{2'} + \phi$ es la mayor cantidad de flujo que se puede enviar de s^2 a t^2.□

6. Algoritmo para obtener un biflujo máximo simétrico

Las anteriores ideas permiten desarrollar el siguiente algoritmo:

```
Algoritmo Biflujo_Máximo_Simétrico(BFMS);
  begin
  Sea G=(V,A); Consideremos el grafo G'(V',A') correspondiente
  para el problema P3a y P3b; Sean las capacidades u' las
  definidas en dicha red; Sea Z^1 = Z^2:=0;
  Obtener mediante la aplicación de cualquier algoritmo de flujo
  máximo la mayor cantidad de flujo que se puede enviar desde
  las fuentes s^1 y h a los sumideros t^1 y h'; Sea Z^1 el
  correspondiente flujo; Sea f^1 la cantidad de flujo enviada de
  s^1 a t^1; f^2 = 0;
  Hacer Z^2 := Z^1; r^1_ij = r^2_ij := u'_ij − z^1_ij + z^1_ji   ∀(i, j) ∈ A';
  Actualizar_f^2(f^1,f^2,R^1,R^2);
  if f^1>f^2 then {En otro caso tenemos el Biflujo Máximo
  Simétrico}
      begin
      Actualizar_f^1(f^1,f^2,R^1,R^2,(f^1-f^2)/2); (caso iii)
      Actualizar_f^2(f^1,f^2,R^1,R^2);
      If f^1>f^2 then Actualizar_f^1(f^1,f^2,R^1,R^2,f^1-f^2); (caso
      iii(a))
      end
  end.
Procedure Actualizar_f^2(f^1,f^2,R^1,R^2);
  begin
  Δ := 2^⌈log 2U⌉;
  while (Δ ≥ 1) and (f^2<f^1) do
    begin
    d^1(t^2) := 0; Obtener d^1 mediante un BFS inverso en R^1
    empezando por t^2;
    d^2(s^2) := 0; Obtener d^2 mediante un BFS inverso en R^2
    empezando por s^2;
    i^1:= s^2; i^2:= t^2;
    while (d^1(s^2) < n) and (d^2(t^2) < n) and (f^2<f^1) do
      begin
```

```
      θ := min(f¹-f², Δ);
      if (i¹ ≠ t²) then
        if (i¹ tiene un arco Δ-admisible) then
        Avanzar(i¹, d¹, R¹, θ, pred¹)
        else Retroceder(i¹, d¹, R¹, θ, pred¹, s²);
      if (i² ≠ s²) then
        if (i² tiene un arco Δ-admisible) then
        Avanzar(i², d², R², θ, pred²)
        else Retroceder(i², d², R², θ, pred², t²);
      if (i¹ = t²) and (i² = s²) then i¹ := s²; i² := t²; f² := f² + θ
      end;
    if (i¹ ≠ s²) then Restablecer(i¹, s², R¹, θ, pred¹);
    if (i² ≠ t²) then Restablecer(i², t², R², θ, pred²);
    Δ := Δ / 2
    end
  end;


Procedure Avanzar(i,d,R,θ,pred);
  begin
  Sea (i,j) un arco Δ-admisible;
  r_ij = r_ij − θ; r_ji = r_ji + θ;
  pred(j) := i; i := j
  end;

Procedure Restablecer(i,s,R,θ,pred);
  begin
  while (i≠s) do
    begin
    j := i; i := pred(i);
    r_ij = r_ij + θ; r_ji = r_ji − θ
    end
  end;
Procedure Retroceder(i,d,R,θ,pred,s);
  begin
  d(i) := min{d(j) + 1 : (i, j) ∈ A'(i) y r_ij ≥ Δ};
  if i ≠ s then
    begin
    j := i; i := pred(i);
    r_ij = r_ij + θ; r_ji = r_ji − θ
    end
  end;

Procedure Actualizar_f¹(f¹, f², R¹, R², φ);
  begin
  Δ := 2^⌈log 2U⌉;
  while (Δ ≥ 1) and (φ > 0) do
    begin
```

```
d¹(s¹) ≔ 0; Obtener d¹ mediante un BFS inverso en R¹
empezando en s¹;
d²(s¹) ≔ 0; Obtener d² mediante un BFS inverso en R²
empezando en s¹;
i¹:= s²; i²:= t¹;
while (d¹(s¹) < n) and (d¹(s¹) < n) and (φ > 0) do
  begin
  θ := min(φ,Δ);
  if (i¹≠s¹) then
    if (i¹ tiene un arco Δ-admisible) then
    Avanzar(i¹,d¹,R¹,θ,pred¹)
    else Retroceder(i¹,d¹,R¹,θ,pred¹,s²);
  if (i²≠s¹) then
    if (i² tiene un arco Δ-admisible) then
    Avanzar(i²,d²,R²,θ ,pred²)
    else Retroceder(i²,d²,R²,θ,pred²,s²);
  if (i¹=s¹) and (i²=s¹) then i¹:=s²;i²:=t¹;f¹:=f¹-θ;φ:=φ-
  θ;
  end;
 Δ ≔ Δ / 2
 end
end;
```

El algoritmo anterior, comienza obteniendo el flujo máximo entre las fuentes s^1, h y los sumideros t^1, h'. Así $f^1 = f^{1*}$ unidades de flujo son enviadas de s^1 a t^1. A continuación, se intentan enviar $f^2 = f^1$ unidades de flujo de s^2 a t^2 en R^1 y de t^2 a s^2 en R^2 (esto se realiza en la primera llamada al procedimiento *Actualizar_*f^2). Esta forma de proceder impide que el caso $f^1 < f^2$ pueda darse. Se observa que, únicamente, se pueden dar los casos i) y iii) del teorema 3.5; nunca el ii). Por lo tanto, después de la llamada a *Actualizar_*f^2, o bien es $f^1 = f^2$ (con lo que se ha obtenido el biflujo máximo simétrico) o bien f^1 sigue siendo mayor que f^2. En este último caso, $\phi = (f^1 - f^2)/2$ unidades de flujo son devueltas desde t^1 a s^1, tanto en R^1 y como en R^2. Esta operación se realiza en la primera llamada a *Actualizar_*f^1. En este punto se intentan enviar esas ϕ unidades de flujo de s^2 a t^2 en R^1 y de t^2 a s^2 en R^2. Esta operación es realizada cuando se llama por segunda vez al procedimiento *Actualizar_*f^2. Si se han enviado las ϕ unidades mencionadas, el

algoritmo termina y $f^1=f^2$; en otro caso, todavía se tiene que $f^1>f^2$ y se envían $\phi=f^1-f^2$ unidades desde t^1 a s^1, tanto en R^1 como en R^2. Esta operación se realiza en la segunda llamada a *Actualizar_*f^1, obteniendo, en su caso, el biflujo máximo simétrico.

Para identificar, simultáneamente, un camino Δ-admisible en cada red residual ($R^k(\Delta)$ con k=1,2,) se procede de la manera siguiente. En cada fase de escalado se calculan las etiquetas distancias exactas d^k, k=1,2. El algoritmo mantiene un índice de predecesor $pred^k(i^k)$ con cada nodo i^k de cada red $R^k(\Delta)$ y realiza iterativamente pasos *Avanzar* o *Retroceder* sobre el último nodo de cada camino admisible parcial. Cada paso *Avanzar* actualiza las capacidades residuales del arco Δ-admisible (i^k,j^k) por $r_{ij}=r_{ij}-\theta$ y $r_{ji}=r_{ji}+\theta$. Si no se identifica un arco Δ-admisible, se realiza un paso *Retroceder* que deshace el envío de θ unidades de flujo anterior. Si se alcanzan los respectivos nodos sumideros, el proceso se repite comenzando en las respectivas fuentes. Por la forma en que se desarrolla el algoritmo es claro que esto únicamente sucede si se han detectado, a la vez, un camino Δ-admisible en R^1 y en R^2, incrementándose f^2 ó decrementándose f^1 en θ unidades de flujo.

Una fase de escalado acaba cuando la etiqueta distancia de al menos un nodo fuente es mayor o igual que n. Si esto ocurre, al menos un i^k puede ser distinto de su nodo fuente, lo que significa que se han enviado θ unidades de flujo a lo largo del camino parcial identificado para la correspondiente red residual. El procedimiento *Restablecer* se encarga de devolver esas θ unidades de flujo. Finalmente, para obtener el patrón de biflujo máximo simétrico se deshace el cambio de variable, es decir:

$$\forall(i,j)\in A,\quad \begin{matrix} z_{ij}^1=\max(0,u'_{ij}-r_{ij}^1) \\ z_{ij}^2=\max(0,u'_{ij}-r_{ij}^2) \end{matrix} \Rightarrow x_{ij}^1=\frac{(z_{ij}^1+z_{ij}^2)}{2}-u_{ij},\ x_{ij}^2=\frac{(z_{ij}^1-z_{ij}^2)}{2}$$

A continuación aplicaremos el algoritmo anterior sobre el ejemplo representado en la Figura 3.4. El proceso para obtener el patrón de biflujo máximo simétrico dado en la Figura 3.5, es mostrado en la Figura 3.6.

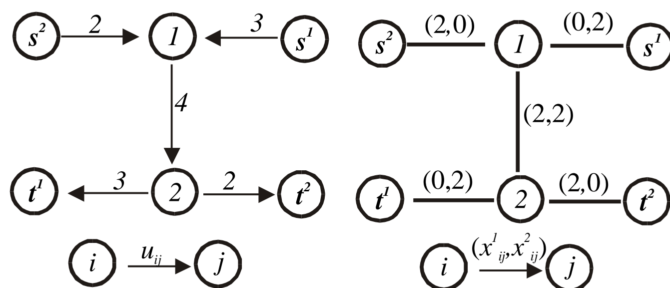


Figura 3.4. *Figura 3.5.*

La Figura 3.6a, representa la red *G'* con las capacidades *u'*. La Figura 3.6b, muestra la red residual R^1 que coincide con R^2, cuando se han enviado $f^1+\delta_h$=9 unidades de flujo desde los nodos s^1, *h* a los nodos t^1, *h'*, donde $f^1=3$ y $f^2=0$. Se puede observar en esta figura que únicamente hay un camino de s^2 a t^2 en R^1 de capacidad 1 y de t^2 a s^2 en R^2 de capacidad 2. Por ello, al llamar por primera vez al procedimiento *Actualizar_*f^2, se identifican ambos caminos cuando el procedimiento se encuentre en la fase Δ=1 (Figura 3.6c). Tras enviar 1 unidad de flujo a través de estos caminos en sus respectivas redes incrementales (Figuras 3.6d y 3.6e), se puede observar que en $R^1(1)$ no hay camino incremental (aunque sí lo hay en $R^2(1)$). En este punto tenemos que $f^1=3$ y $f^2=1$. Por lo tanto, $f^1>f^2$, lo que implica que $\phi=(f^1-f^2)/2=1$. Entonces tenemos que el procedimiento *Actualizar_*f^1 envía 1 unidad de flujo de t^1 a s^1, tanto en R^1 como en R^2. El resultado de esta operación se muestra en las Figuras 3.6f y 3.6g respectivamente.

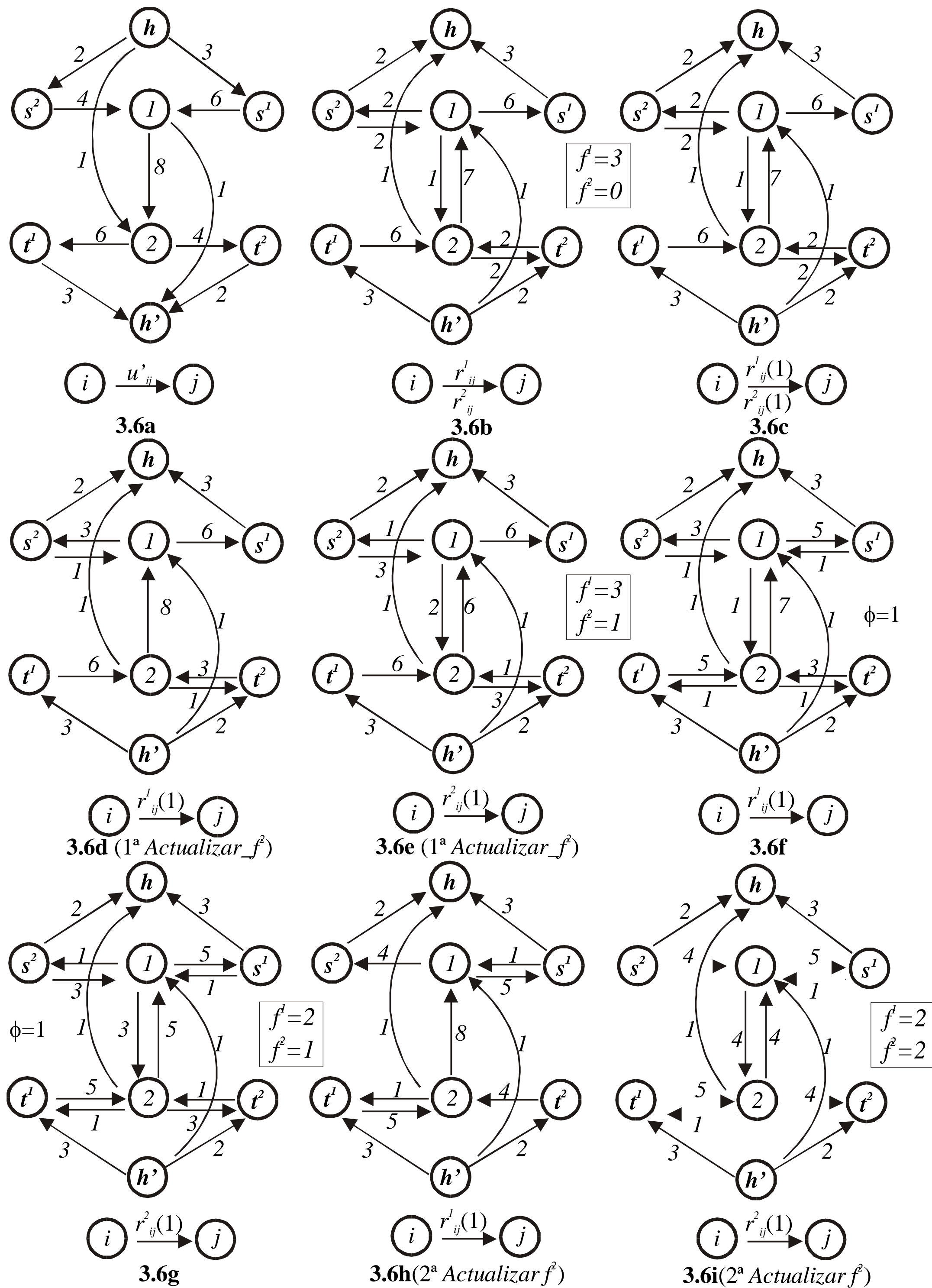


Figura 3.6. Traza del algoritmo sobre el ejemplo

Ahora bien, tenemos que $f^1 = 2$ y $f^2 = 1$, pero lo importante es que podemos enviar 1 unidad de flujo de s^2 a t^2 en R^1 y de t^2 a s^2 en R^2. Esta operación se realiza cuando se llama por segunda vez al procedimiento *Actualizar_*f^2. El resultado de esta operación se muestra en las Figuras 3.6h y 3.6i de donde se tiene que $f^1 = 2$ y

$f^2 = 2$. En este punto no es necesario llamar al procedimiento *Actualizar_*f^1. Finalmente, el patrón de biflujo máximo simétrico se obtiene deshaciendo los cambios de variables (Figura 3.5).

6.2 Complejidad del algoritmo

En esta sección demostraremos que la complejidad del algoritmo, en el caso peor, es O($nmlogU$).

También en esta ocasión, la primera operación que realiza el algoritmo es la de obtener el flujo máximo entre las fuentes s^1, h y los sumideros t^1, h' y puede ser efectuada por cualquier algoritmo de flujo máximo. Bastaría entonces con elegir uno con una complejidad teórica inferior o similar a la correspondiente de las siguientes operaciones. El algoritmo presentado ejecuta al menos una vez, y a lo sumo dos veces, el procedimiento *Actualizar_*f^2. El procedimiento *Actualizar_*f^1 se ejecuta, como máximo, dos veces. Ambos procedimientos requieren el mismo esfuerzo computacional. Cada uno de ellos realiza $\lceil \log 2U \rceil$ fases de escalado. Cada fase de escalado Δ es llevada a cabo hasta que alguna de las etiquetas distancia, en R^1 o en R^2, de las correspondientes fuentes, sea mayor o igual que n.

Lema 3.5. *Cada fase de escalado de Actualizar_*f^1 *o de Actualizar_*f^2 *requiere un esfuerzo computacional de O(*nm*).*

Demostración. Como ambos procedimientos tienen la misma complejidad, calculemos la complejidad del procedimiento *Actualizar_*f^2. Supongamos que nos fijamos en la etiqueta distancia $d^1(s^2)$ en R^1 y que esta es la que determina la condición de parada de cada fase de escalado cuando $d^1(s^2) \geq n$. Utilizando las mismos cálculos desarrollados en el lema 3.3, la complejidad de una fase, considerando que la etiqueta distancia $d^1(s^2)$ en R^1 determina la condición de parada, es O(nm).

La misma complejidad en el caso peor se obtendría si se considerara que es $d^2(t^2)$ la etiqueta distancia en R^2 la que determina la condición de parada. Como, en cada fase, alguna de las dos condiciones debe ocurrir, la complejidad de cada una de las fases coincide con el máximo de las complejidades de ambas. Por lo tanto, la complejidad es $O(nm)$.□

Teorema 3.6. *El algoritmo tiene una complejidad* $O(nm \log U)$.

Demostración. Por el lema 3.5, cada fase de escalado emplea un tiempo de $O(nm)$ y, como el número de fases de escalado es $\lceil \log 2U \rceil$, obtenemos que cada procedimiento requiere un tiempo $O(nmlogU)$. Como el algoritmo llama, como máximo, cuatro veces a los mencionados procedimientos, entonces se ejecuta en un tiempo $O(nmlogU)$.□

7. El problema de biflujo máximo biobjetivo

La formulación del problema de biflujo máximo (3.1), permite de forma natural formalizar el problema de biflujo máximo biobjetivo, es decir:

$$\text{Max} \quad f^1$$
$$\text{Max} \quad f^2$$
$$\sum_{j \in Suc(i)} x_{ij}^k - \sum_{j \in Pred(i)} x_{ji}^k = \begin{cases} f^k & si \;\; i = s^k \\ 0 & i \in V - \{s^k, t^k\} \\ -f^k & si \;\; i = t^k \end{cases} \qquad k = 1,2 \qquad (3.4a)$$

$$\left| x_{ij}^1 \right| + \left| x_{ij}^2 \right| \le u_{ij}, \; (i,j) \in A \qquad (3.4b)$$

Cualquier biflujo x es denominado una *solución factible*. El conjunto de soluciones factibles o *espacio de decisiones* es denotado por X. Si definimos la función $f(x) = (f^1(x), f^2(x))$, entonces la imagen de X mediante f es $f(X) = \{(f^1(x), f^2(x)) \,/\, x \in X\}$. $f(X)$ recibe el nombre de *espacio objetivo*.

Para k=1,2, sea x^{k*} la solución del problema que considera como objetivo únicamente el valor de $f^k(x^{k*})=f^{k*}$, es decir el problema de flujo máximo de un único bien asociado con el correspondiente valor de k. En general, no hay un biflujo $x^*=(x^{1*},x^{2*})$ tal que $f(x^*)=(f^{1*},f^{2*})=f^*$. Por esta razón a f^* se le denomina punto ideal o utopía.

La solución del problema de biflujo máximo biobjetivo es elegida de entre el conjunto de soluciones eficientes, es decir, soluciones que satisfacen la siguiente definición:

Definición 3.2. Una solución factible $x \in X$ del problema de biflujo máximo biobjetivo es eficiente si y sólo si, no existe otra solución factible $x' \in X$ tal que $f^k(x') \geq f^k(x)$ para k=1,2, con $f^k(x') \neq f^k(x)$ para al menos un k.

Realmente, cuando obtenemos x^{k*} para un valor de k fijo, hemos de obtener un biflujo, es decir, por ejemplo para k=1 obtenemos $x_1^*=(x^{1*},x^{2'})$. x_1^* es un biflujo tal que para el primer objetivo se obtiene un flujo máximo, y para el segundo objetivo se envía la mayor cantidad de flujo posible manteniendo el valor alcanzado para el primer objetivo que denotaremos por $(f^{1*},f^{2'})$. Este problema corresponde al biflujo máximo lexicográfico donde los objetivos son considerados en el orden (f^1,f^2). Así, $x_2^*=(x^{1'},x^{2*})$ con imagen $(f^{1'},f^{2*})$ es la solución óptima del problema de biflujo máximo lexicográfico con los objetivos en el orden (f^2,f^1). Es evidente que, en general, x_1^* y x_2^* son biflujos eficientes. Es obvio, que el algoritmo dado en la sección (4) obtiene x_1^*. Para obtener, x_2^* bastaría utilizar el mismo algoritmo intercambiando las fuentes y sumideros, es decir, $s^1 \leftrightarrow s^2$ y $t^1 \leftrightarrow t^2$; y obtener el patrón de biflujo de la siguiente manera:

$$\forall (i,j) \in A, \quad \begin{array}{l} z_{ij}^1 = \max(0, u'_{ij} - r_{ij}^1) \\ z_{ij}^2 = \max(0, u'_{ij} - r_{ij}^2) \end{array} \Rightarrow x_{ij}^1 = \frac{(z_{ij}^1 - z_{ij}^2)}{2}, \; x_{ij}^{2*} = \frac{(z_{ij}^1 + z_{ij}^2)}{2} - u_{ij}$$

es decir, como si el cambio de variable que se ha considerado fuese el anterior.

Nuestro objetivo en esta sección es caracterizar el conjunto de biflujos eficientes en el espacio objetivo, es decir, $E[f(X)]$. En nuestro caso, X es un poliedro compacto, por lo tanto $f(X)$ es un poliedro compacto también. Esto permite caracterizar a $E[f(X)]$ mediante el conjunto de *puntos eficientes extremos* en el espacio objetivo $E_{ex}[f(X)]$. Con el fin de caracterizar a $E_{ex}[f(X)]$ daremos los siguientes teoremas.

Teorema 3.7. *Un biflujo* $x \in X$ *es eficiente si y sólo si* $(f^1(x), f^2(x)) = (f^1, f^2)$ *es tal que* $f^1 + f^2 = f^{1*} + f^{2'}$.

Demostración. Por el teorema de biflujo-máximo corte-mínimo existe un corte $[S, \overline{S}]$ tal que $f^{1*} + f^{2'} = u^*[S, \overline{S}]$. Evidentemente, cualquier otra solución eficiente (f^1, f^2) distinta de $(f^{1*}, f^{2'})$ debe cumplir que $f^1 < f^{1*}$ y $f^2 > f^{2'}$. Como $f^{1*} + f^{2'}$ es máximo, únicamente tenemos que descartar el caso $f^1 + f^2 < f^{1*} + f^{2'}$ para finalizar la demostración. Supongamos entonces que $f^1 + f^2 < f^{1*} + f^{2'}$. Por el teorema de biflujo-máximo corte-mínimo podemos concluir que el biflujo $(f^{1*} + f^{2'} - f^2, f^2)$ es factible . Entonces $f^1 < f^{1*} + f^{2'} - f^2$ y, por lo tanto, (f^1, f^2) no es eficiente, lo que contradice la suposición. En definitiva, se ha de tener que $f^1 + f^2 = f^{1*} + f^{2'}$. □

En particular, se observa que los puntos eficientes extremos $(f^{1*}, f^{2'})$ y $(f^{1'}, f^{2*})$ son tales que $f^{1*} + f^{2'} = f^{1'} + f^{2*}$.

Teorema 3.8. *Únicamente se puede dar uno de los dos siguientes casos sobre el conjunto de puntos extremos eficientes del problema de biflujo máximo biobjetivo*:

(i) $E_{ex}[f(X)] = f^*$ (*El punto ideal*)

(ii) $E_{ex}[f(X)] = \{(f^{1*}, f^{2'}), (f^{1'}, f^{2*})\}$ (*Soluciones lexicográficas*)

Demostración. Si ocurre el caso i) es por que el punto ideal es factible y por lo tanto este es único, ya que no existe ningún punto extremo eficiente tal que $f^1 < f^{1*}$ y $f^2 > f^{2*}$ o *viceversa*. En el caso de que el punto ideal no sea factible, tenemos como puntos extremos eficientes $(f^{1*}, f^{2'})$ y $(f^{1'}, f^{2*})$. Por el teorema 3.7, cualquier solución eficiente pertenecerá a la recta definida por los dos puntos anteriores. Por lo tanto, estos son los únicos puntos extremos eficientes.□

8. Resultados computacionales

En esta sección reflejaremos los resultados obtenidos en la implementación del algoritmo de biflujo máximo presentado en la sección 4. Denotaremos por BFMP a la codificación en Pascal estándar del mencionado algoritmo. BFMP ha sido ejecutado en una estación de trabajo 715/80 HP9000.

Las complejidades teóricas de los algoritmos de flujos son funciones del número de nodos n, número de arcos m, y de la máxima capacidad de los arcos U. Por lo tanto, los casos particulares del problema de biflujo máximo son redes generadas aleatoriamente con diferentes tamaños de esos parámetros. Los problemas test fueron generados usando NETGEN [52] y MFGEN (ver capítulo 2).

Para ambos generadores, la especificación de un caso particular viene dada por los siguientes cuatro parámetros: *semilla, n, m,* y *U*. Dada cualquier red aleatoria, los nodos fuentes y sumideros han sido los siguientes: $s^1 = 1$, $s^2 = [n/4]$, $t^1 = n$ y $t^2 = [3n/4]$. En la Tabla 3.2 se muestran los valores usados para n, m y el ratio m/n. Debemos notar que una secuencia idéntica del ratio se realiza para cada valor de n. Así, dado n y el ratio, se calcula el valor de m.

n	m/n	m
200	10, 30, 50	2000, 6000, 10000
500	10, 30, 50	5000, 15000, 25000

800	10, 30, 50	8000, 24000, 40000

Tabla 3.2. Número de nodos, ratio y arcos

Los valores para *U* han sido 1, 100, y 10.000. Entonces, el número de posible combinaciones de los parámetros es 3x3x3=27. Para cada una de estas especificaciones se han considerado 5 réplicas utilizando las siguientes 5 semillas: 12345678, 36581249, 23456183, 46545174 y 35826749. Cada una de estas especificaciones han sido la entrada de los dos generadores. Por lo tanto, se han resuelto un total de 270 casos particulares del problema de biflujo máximo.

En el experimento computacional se ha medido, en segundos, el tiempo de CPU. Este tiempo no incluye el tiempo empleado en las operaciones de entrada/salida. También, consideramos el tiempo de CPU (CPU_INI) consumido en la obtención del flujo máximo entre los nodos fuentes $\{s^1, h\}$ y los nodos sumideros $\{t^1, h'\}$. Para esta operación, hemos utilizado el algoritmo de preflujo propuesto por Goldberg [38], con la regla que selecciona el nodo activo con mayor etiqueta distancia. Recordemos que esta regla da lugar a un algoritmo de complejidad O($n^2\sqrt{m}$) (ver Cheriyan y Maheswari [19]). Además, del capítulo anterior extraemos que este algoritmo es el más rápido para los generadores que hemos utilizado (ver Derigs y Meier [24], Ahuja et al. [2] y Sedeño-Noda et al. [67]).

En la implementación del algoritmo de biflujo máximo se han incorporado las estrategias para reducir el número de actualizaciones de la etiqueta distancia debidas a Ahuja y Orlin [5] y a Derigs y Meier [24] que fueron comentados en el capítulo 2.

Las Tablas 3.3 y 3.4 muestran los resultados obtenidos para los problemas generados en las siguientes columnas: media del tiempo de CPU, media del tiempo de CPU inicial, y la media para los valores de f^1 y f^2. Los grupos para calcular la estadística han sido indexados mediante los parámetros *n* y *m*. A partir de estas tablas, podemos concluir que el tiempo de CPU inicial (CPU_INI) es, aproximadamente, el 22 por ciento del tiempo de CPU total. Además, este tiempo de CPU total es mayor en los problemas generados por NETGEN que en aquellos creados por MFGEN. Sin embargo, hay que tener en cuenta que ambos generadores, con parámetros idénticos, construyen problemas con diferentes

particularidades. Por ejemplo, el valor del biflujo en las redes NETGEN es mayor que en las redes MFGEN.

n	*m*	CPU_INI	CPU	F1	F2
200	2000	0,121	0,251	43387,533	72782,733
200	6000	0,087	0,353	65553,000	126234,533
200	10000	0,097	0,495	98485,867	180866,333
500	5000	0,469	0,904	63860,667	126011,200
500	15000	0,236	1,535	99038,067	172121,267
500	25000	0,313	2,647	124986,867	214007,933
800	8000	0,566	1,186	44279,800	91050,200
800	24000	0,391	2,418	78920,933	143220,133
800	40000	0,442	2,767	112804,733	215003,400

Tabla 3.3. Resultados para NETGEN

n	*m*	CPU_INI	CPU	F1	F2
200	2000	0,051	0,137	25824,400	32703,733
200	6000	0,065	0,337	95513,867	101252,667
200	10000	0,085	0,602	164744,000	164111,533
500	5000	0,148	0,416	27480,867	34135,133
500	15000	0,195	0,792	95967,133	95384,600
500	25000	0,225	1,279	163336,600	157469,867
800	8000	0,234	0,649	29076,933	31020,267
800	24000	0,306	1,198	93394,267	91897,467
800	40000	0,333	2,051	158336,533	160736,000

Tabla 3.4. Resultados para MFGEN

En la Figura 3.7 se muestra el tiempo de CPU del algoritmo con respecto al producto *n*x*m* para ambos generadores. Hay que notar que, aunque los tiempos de CPU son mayores siempre para los problemas obtenidos mediante NETGEN, el ratio de crecimiento es similar.

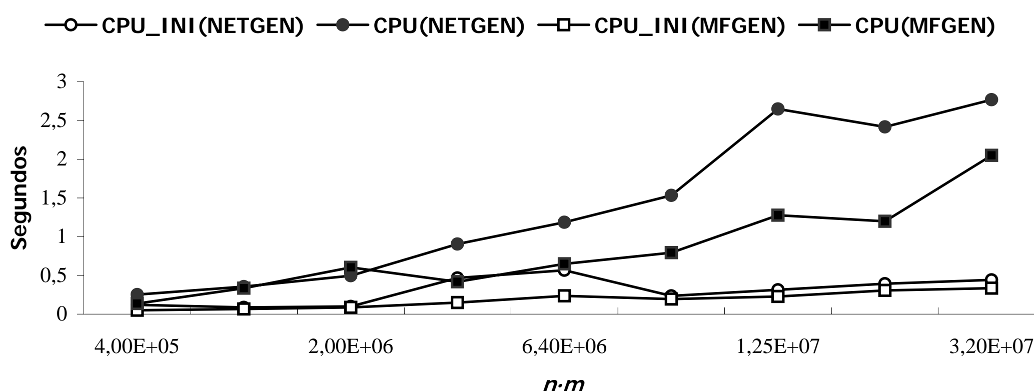


Figura 3.7. Tiempo de CPU para ambos generadores.

En la Figura 3.8, se muestra la relación entre el tiempo de CPU y la suma $f^1 + f^2$. Observamos que no hay una clara correlación entre estas variables. Además, hemos observado que la capacidad

máxima no tiene una influencia significativa sobre el tiempo de CPU. Estas dos últimas afirmaciones están relacionadas con el carácter polinomial de la complejidad teórica del algoritmo.

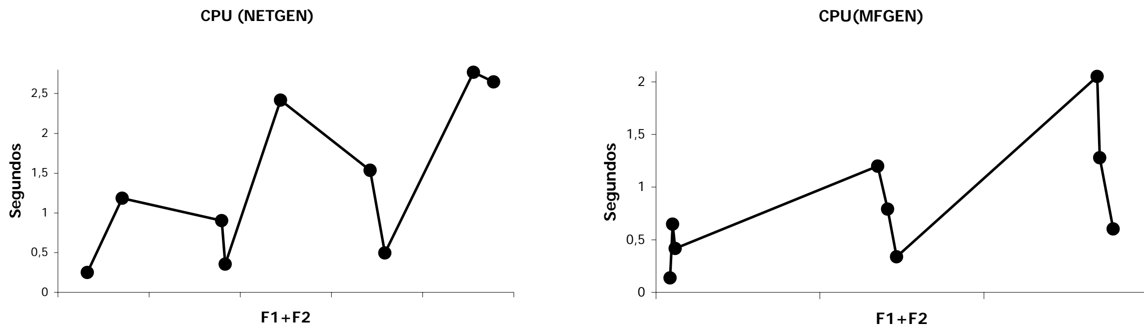
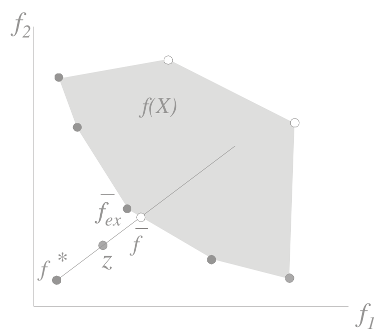


Figura 3.8. Tiempo de CPU contra $f^1 + f^2$.



Capítulo 4

Algoritmos para el problema de Flujo de Coste Mínimo Biobjetivo

1. Introducción

En determinados problemas de Optimización Combinatoria la selección de la mejor solución debe realizarse teniendo en cuenta más de un criterio. Estas situaciones también se dan en los problemas de flujos en redes, donde los criterios a considerar podrían ser la minimización de la longitud de las rutas seleccionadas, la minimización del tiempo de llegada a los puntos destinos, la minimización del deterioro de los bienes que circulan por la red, la minimización de la capacidad de carga no usada en los vehículos seleccionados, la maximización de la seguridad, fiabilidad, etc..

Los problemas de flujo de coste mínimo multiobjetivo ya han merecido la atención de diversos autores. Destacamos los siguientes trabajos: Aneja y Nair [9] estudian el caso particular del problema de transporte bicriterio desarrollando un procedimiento capaz de generar el conjunto de puntos extremos eficientes en el espacio objetivo. Calvete y Mateo [16] introducen un método lexicográfico para los problemas de flujos en redes multiobjetivo con prioridades anticipidadas (*pre-emptive priorities*). Isermann [45] estudia el problema de transporte multicriterio desarrollando un procedimiento para identificar la región eficiente en el espacio de decisiones, tomando como base las propiedades de adyacencia entre puntos extremos. Mustafa y Goh [60] desarrollan un método para ajustar de manera entera soluciones de compromiso no enteras obtenidas mediante DINAS, resolviendo interactivamente problemas de flujos en redes bicriterio y tricriterio. Klingman y Mote [51] realizan una adaptación del método Simplex multiobjetivo de Yu y Zeleny [76] al caso de redes multiobjetivo. Lee y Pulat [53] desarrollan un método para determinar el conjunto de soluciones eficientes extremas en el espacio objetivo para el problema de flujo de coste mínimo biobjetivo. Posteriormente, Lee y Pulat [54] consideran el problema de flujos en redes bicriterio donde las variables de flujo están restringidas a tomar valores enteros. Su procedimiento utiliza soluciones eficientes de la versión continua del problema de flujos en redes bicriterio. Malhotra y Puri [56] introducen un método para obtener todos los puntos eficientes

en el espacio objetivo del problema de flujos en redes bicriterio, el cual utiliza una versión del método Out-of-Kilter. Pulat et al. [62] describen un método de Flujos en Redes Bicriterio basado en análisis paramétrico para obtener todas las bases eficientes en el espacio de decisiones. Finalmente, Ringuest y Rinks [63] desarrollan un método interactivo para el problema de transporte multicriterio. Una clasificación de estos problemas puede ser consultada en Current y Min [22].

En este capítulo nos dedicamos al problema de flujo de coste mínimo, considerando dos objetivos simultáneamente. En este caso la optimalidad habrá que buscarla en un conjunto de soluciones factibles denominadas soluciones eficientes. Nuestra intención es caracterizar a este conjunto en el espacio objetivo.

A partir de la teoría de Programación Lineal Multiobjetivo (ver por ejemplo Steuer[74], Yu[75], Yu y Zeleny[76]) se conoce que el conjunto de puntos extremos eficientes en el espacio de decisiones puede ser obtenido mediante métodos de programación paramétrica (ver, por ejemplo, Gass y Saaty [33], Geoffrion [34], Gal [32], Bryson [14] y el ya mencionado método de Pulat et al. [62]). Sin embargo, no hemos encontrado en la literatura un método de programación paramétrica que determine todas las soluciones extremas eficientes en el espacio objetivo.

El cálculo del conjunto de puntos extremos eficientes en el espacio objetivo es, desde un punto de vista computacional, muy atractivo, ya que el máximo número de puntos extremos eficientes adyacentes a otro es la dimensión del espacio objetivo. Sin embargo, en el espacio de decisiones este número es a lo sumo $m - n + 1$, donde n es el número de nodos y m es el número de arcos en la red. En el caso que nos ocupa, es claro que, normalmente 2 es menor que $m - n + 1$. Por otro lado, una vez obtenidos los puntos extremos eficientes en el espacio objetivo, en una segunda fase es posible determinar todas las soluciones eficientes en el espacio de decisiones debido a las propiedades de convexidad del problema de flujo de coste mínimo biobjetivo.

El problema en estudio puede requerir, que los flujos sean enteros (caso entero). A la hora de diseñar algoritmos para este

nuevo problema, hemos de tener en cuenta si se contempla la mencionada restricción.

Presentamos dos nuevos algoritmos para la resolución del caso continuo. Estos algoritmos son muy diferentes pero justificaremos la conveniencia de ambos, aún cuando uno de ellos es extremadamente más rápido que el otro. Dicho sea de paso, uno de los algoritmos presentados es el que emplea menos tiempo de CPU de entre todos los métodos diseñados para resolver el mencionado problema.

Presentamos un algoritmo para resolver el caso entero. Queremos destacar que este algoritmo es el único existente en la actualidad que resuelve dicho problema utilizando la estructura subyacente de red. Un intento para resolver este caso es debido a Lee y Pulat [54], quienes plantean un procedimiento que, como mostraremos, no calcula todo el conjunto de soluciones eficientes.

2. Preliminares y formalización del problema

Dada una red dirigida $G = (V, A)$, sea $V = \{1,\dots,n\}$ el conjunto de nodos y A el conjunto de arcos. Para cada $i \in V$, sea el entero b_i la oferta/demanda del nodo i y para cada arco $(i,j) \in A$ sean u_{ij} y l_{ij} respectivamente la cota superior e inferior del flujo que puede circular a través del arco (i,j). Denotamos por c_{ij}^k el coste por unidad de flujo sobre el arco (i,j) en la k-ésima función objetivo, $k=1,2$.

Si x_{ij} es la cantidad de flujo que ha de circular por (i,j), $Suc(i) = \{j \in V / (i,j) \in A\}$ y $Pred(i) = \{j \in V / (j,i) \in A\}$, el problema de flujos enteros en redes biobjetivo (FERB) puede ser formalizado de la siguiente manera:

$$\begin{aligned}
&\text{Min} \quad f_1(x) = \sum_{i\in V}\ \sum_{j\in Suc(i)} c_{ij}^1 x_{ij} \\
&\text{Min} \quad f_2(x) = \sum_{i\in V}\ \sum_{j\in Suc(i)} c_{ij}^2 x_{ij} \\
&s.a: \\
&\qquad \sum_{j\in Suc(i)} x_{ij} - \sum_{j\in Pred(i)} x_{ji} = b_i, \forall i \in V \qquad (4.1a) \\
&\qquad l_{ij} \le x_{ij} \le u_{ij}, \forall (i,j) \in A \qquad (4.1b) \\
&\qquad x_{ij} \text{ entero} \qquad (4.1c)
\end{aligned}$$

El problema de flujos en redes biobjetivo (FRB) viene definido por (4.1a)-(4.1b). Cualquier vector x que satisface (4.1a)-(4.1b) es denominado *solución factible* del problema FRB. El conjunto de soluciones factibles o *espacio de decisiones* se denota por X y su imagen a través de $f(X) = \{(f_1(x), f_2(x)) / x \in X\}$ es denominada *espacio objetivo*. Así, denotaremos por X^I al espacio de decisiones del problema FERB y por $f(X^I)$ al espacio objetivo del mismo problema.

Dado el problema FRB, sea x_k^* la solución del siguiente problema:

$$\begin{aligned}
&f_k^* = f_k^*(x_k^*) = \min \sum_{i\in V}\ \sum_{j\in Suc(i)} c_{ij}^k x_{ij} \quad k = 1,2 \qquad (4.2) \\
&\qquad \text{s.a}: \quad (4.1a) \text{ y } (4.1b)
\end{aligned},$$

es decir, para $k = 1,2$, f_k^* es el valor óptimo del correspondiente problema de flujos uniobjetivo.

En general, no existe una única solución factible del problema FRB que minimice los dos objetivos de manera simultánea. En otras palabras, no existe una solución óptima global. Lo mismo ocurre para el problema FERB. Por lo tanto, el vector $f^* = (f_1^*, f_2^*)$ es denominado *punto ideal* o utopía. Debido a esta dificultad, las soluciones del problema FRB son buscadas entre el conjunto de puntos eficientes, soluciones no dominadas u óptimos de Pareto, es decir, puntos que satisfacen las siguientes definiciones:

Definición 4.1: *Una solución factible $x \in X$ del problema FRB es eficiente sí, y sólo sí, no existe otra solución factible $x' \in X$ tal que $f(x') \le f(x)$ y $f(x') \neq f(x)$.*

Definición 4.2: *Una solución factible* $x \in X^I$ *del problema FERB es eficiente sí, y sólo sí, no existe otra solución factible* $x' \in X^I$ *tal que* $f(x') \leq f(x)$ *y* $f(x') \neq f(x)$.

Denotaremos por $E[X]$ al conjunto de soluciones eficientes de X. Por extensión, $E[f(X)] = \{f(x)/x \in E[X]\}$ es denominado conjunto de soluciones eficientes de $f(X)$. De la misma manera, denotaremos por $E[X^I]$ al conjunto de soluciones eficientes de X^I y por $E[f(X^I)] = \{f(x)/x \in E[X^I]\}$ al conjunto de soluciones eficientes de $f(X^I)$.

La resolución del problema FRB debe llevarse a cabo seleccionando adecuadamente soluciones de los conjuntos $E[X]$ o $E[f(X)]$. Si dicha selección estuviera basada en el conocimiento explícito de dichos conjuntos, el correspondiente proceso podría resultar muy costoso debido a que los correspondientes cardinales pueden ser muy grandes. Sin embargo, dadas las características de X y $f(X)$ en este caso, tanto $E[X]$ como $E[f(X)]$ pueden ser generados a partir de subconjuntos finitos de sus puntos: *Los puntos extremos eficientes*. Sea $E_{ex}[X]$ el conjunto de puntos extremos eficientes de X y sea $E_{ex}[f(X)]$ el correspondiente de $f(X)$. Nos centraremos en identificar $E_{ex}[f(X)]$ y, a través de este conjunto, $E_{ex}[X]$.

Denotamos por $\gamma(x^i, x^j)$ la arista del poliedro compacto X que conecta los puntos extremos x^i y x^j.

Definición 4.3: *Sea* $\gamma(x^i, x^j)$ *una arista que conecta dos puntos extremos eficientes adyacentes* x^i *y* x^j *en* X*. Entonces* $\gamma(x^i, x^j)$ *es denominada arista eficiente.*

Es conocido que los puntos de $E_{ex}[f(X)]$ se corresponden con puntos enteros de X. Sin embargo, no ocurre lo mismo con todos los puntos sobre la frontera eficiente de $f(X)$, aunque algunos si pueden corresponder a valores enteros. Denotaremos por $E_{nex}[X]$ al conjunto de puntos eficientes enteros no extremos que están sobre la frontera eficiente de X y por $E_{nex}[f(X)]$ al correspondiente conjunto en el espacio objetivo.

En el problema FERB, la consideración de la restricción de integridad en las variables de flujo implica que puedan existir puntos eficientes que no están sobre la frontera eficiente de $f(X)$. Estos puntos habrán de ser generados de manera separada. Nuestra intención es calcular todos los puntos eficientes del problema FERB, es decir, $E[f(X^I)]$.

Es evidente que $E_{ex}[f(X)] \subseteq E[f(X^I)]$ y que $E_{nex}[f(X)] \subset E[f(X^I)]$. Así, nuestro propósito es, primero, determinar $E_{ex}[f(X)] \cup E_{nex}[f(X)]$ mediante la resolución del problema FRB y, a continuación, determinar $E[f(X^I)] - (E_{ex}[f(X)] \cup E_{nex}[f(X)])$. Por lo tanto, en primer lugar hemos de diseñar un método para resolver el problema FRB y, posteriormente, mediante este, obtener un algoritmo para resolver el problema FERB.

2.1 Caracterización de soluciones eficientes

El siguiente problema uniobjetivo puede ser utilizado para obtener soluciones eficientes:

$$\begin{array}{ll} \min & d_\pi(f^*, f(x)) \quad (4.3) \\ s.a: & x \in X \end{array} \qquad (P_\pi),$$

donde X es el espacio de decisiones del problema FRB y d_π una métrica apropiada ponderada por el parámetro de pesos π (ver, por ejemplo González-Martín [41] y Steuer [74]). El problema P_π requiere encontrar la solución factible más cercana según la distancia utilizada al punto ideal.

Los dos algoritmos que presentaremos en este capítulo utilizan distintas métricas. El primero de ellos utiliza la distancia lineal ponderada y el segundo la distancia del máximo ponderado (métrica de Tchebycheff).

3. Estudio para la distancia lineal ponderada

Para el caso lineal tenemos que la distancia viene dada por:

$$d_\pi(f^*, f(x)) = \sum_{k=1,2} \pi_k \left| f_k(x) - f_k^* \right| = \sum_{k=1,2} \pi_k \left(f_k(x) - f_k^* \right)$$

Por lo tanto, para obtener soluciones eficientes hemos de resolver el siguiente problema uniobjetivo que denominaremos $P(\lambda)$[1].

$$\begin{array}{ll} \min & \lambda f_1(x) + (1-\lambda) f_2(x) \\ s.a: & x \in X \end{array}$$

donde $0 \leq \lambda \leq 1$ (ver Geoffrion [34]).

Sea S_λ el conjunto de soluciones de $P(\lambda)$. Definimos $f(S_\lambda) = \{f(x) / x \in S_\lambda\}$. De Geoffrion [34] se pueden extraer los siguientes resultados:

Teorema 4.1.

(i) $\forall x \in E[X], \exists \lambda \in [0,1]$ *tal que* $x \in S_\lambda$

(ii) *Si* $\lambda \in (0,1)$ *entonces* $S_\lambda \subseteq E[X]$. *Si* $\lambda=0$, *entonces* $x_2^* \in S_0$, *y si* $\lambda=1$, $x_1^* \in S_1$.

El teorema anterior establece una correspondencia entre el conjunto $E[X]$ y el intervalo [0,1] a través de $P(\lambda)$. El resultado siguiente permite afirmar que S_λ contiene un único punto extremo eficiente de $E[X]$ o está determinado por una arista eficiente de este poliedro.

Teorema 4.2. *Si* λ *es tal que* $0 < \lambda < 1$, *entonces:* i) S_λ *es un conjunto unitario o* ii) $\exists x^1, x^2 \in E[X]$ *con* $x^1 \neq x^2$, *tales que* $S_\lambda = \{\alpha x^1 + (1-\alpha) x^2 / \alpha \in [0,1]\}$.

---

[1] *EN EL CASO LINEAL PREFERIMOS DENOTAR POR* λ *A LOS PESOS* π.

Los anteriores resultados permiten determinar si la solución del problema $P(\lambda)$, fijado λ, es eficiente o no. La cuestión ahora es cómo se ha de variar el parámetro λ para obtener el conjunto de soluciones extremas eficientes. Para ello basta con recordar que el problema $P(\lambda)$ puede considerarse como un problema de programación paramétrica que permite resolver nuestro problema bicriterio (ver Geoffrion [34]).

Presentamos un procedimiento que toma como base el método de Lee y Pulat [53]. Nuestro método es esencialmente un método Simplex para redes biobjetivo. Comienza con el punto extremo eficiente en el espacio objetivo $f(x^0)$ que resulta de resolver $P(\lambda^0)$ con $\lambda^0=1$. Entonces calcula los restantes puntos extremos eficientes en el mencionado espacio mediante una secuencia finita de pivoteos.

Para computar x^0 optimizamos únicamente sobre el primer objetivo. De esta manera obtenemos la estructura de árbol generador fuertemente factible (B^0,L^0,U^0) y los potenciales de los nodos con respecto al primer objetivo π^1 (ver, por ejemplo Ahuja et al. [4] y Cunningham [20]). Sea $\overline{c}_{ij}^1 = c_{ij}^1 - \pi_i^1 + \pi_j^1$ para todo $(i,j) \in A$. Entonces (B^0,L^0,U^0) es óptima y cumple las siguientes condiciones de optimalidad respecto del primer objetivo:

$$\begin{array}{ll} l_{ij} \le x_{ij}^0 \le u_{ij} \text{ y } \overline{c}_{ij}^1 = 0, & \forall (i,j) \in B^0 \\ x_{ij}^0 = l_{ij} \text{ y } \overline{c}_{ij}^1 \ge 0, & \forall (i,j) \in L^0 \\ x_{ij}^0 = u_{ij} \text{ y } \overline{c}_{ij}^1 \le 0, & \forall (i,j) \in U^0 \end{array}$$

Una vez obtenido (B^0,L^0,U^0) y π^1, podemos obtener los potenciales de los nodos π^2 con respecto al segundo objetivo resolviendo el sistema de ecuaciones:

$$\overline{c}_{ij}^2 = c_{ij}^2 - \pi_i^2 + \pi_j^2 = 0, \ \forall (i,j) \in B^0$$

Como x^0 es un punto extremo eficiente en el espacio de decisiones y $f(x^0)$ es punto extremo eficiente en el espacio objetivo, el siguiente paso consiste en encontrar un punto extremo eficiente en el espacio objetivo adyacente a $f(x^0)$.

Antes de dar una definición de adyacencia entre puntos extremos en el espacio objetivo, recordaremos algunos teoremas y

definiciones conocidos para problemas de programación lineal multiobjetivo.

Teorema 4.3. *Sea* $\{x^0,x^1,\ldots,x^i\}$ el *conjunto de todos los puntos extremos eficientes del problema FRB con* $i \geq 2$. *Entonces, empezando con cualquier* x^j $(0 \leq j \leq i)$ *y moviéndonos a los adyacentes, podemos generar el conjunto de todos los puntos extremos eficientes.*

Por lo tanto, encontrar un punto extremo eficiente adyacente a otro en el espacio de decisiones es equivalente a determinar la arista eficiente que los conecta, es decir, elegir una variable no básica adecuada para entrar en la base actual. En ausencia de degeneración, esta arista eficiente puede encontrarse chequeando los costes reducidos $\overline{c}_{ij}^1$ y $\overline{c}_{ij}^2$ correspondientes a todas las variables no básicas en uno de los puntos extremos.

Si la solución actual es degenerada, es decir, tiene más de una base asociada, interesa encontrar aquella que es eficiente. Una definición de base eficiente es la siguiente:

Definición 4.4. *Sea B una base correspondiente a un punto extremo eficiente* x^i *y sea* x^j *un punto extremo eficiente adyacente. Si, en referencia a B, se puede obtener* x^j *a partir de* x^i *mediante un pivoteo, entonces B es llamada base eficiente.*

Sabemos que $f(E_{ex}[X]) \subseteq E_{ex}[f(X)]$, pudiendo ser el contenido estricto. Sin embargo, la adyacencia de puntos extremos eficientes en X no se conserva a través de f. Por lo tanto, hay que introducir el concepto de adyacencia de puntos extremos eficientes en el espacio objetivo:

Definición 4.5. *Dados los puntos extremos eficientes adyacentes* x^i, $x^{i+1},\ldots, x^j$ *en el espacio de decisiones. Entonces* $f(x^i)$ *y* $f(x^j)$ *son puntos extremos eficientes adyacentes si en la secuencia* $f(x^i), f(x^{i+1}),\ldots, f(x^j)$, *los únicos puntos extremos de* $f(X)$ *son* $f(x^i)$ *y* $f(x^j)$.

Nota: A partir de las definiciones 4.3 y 4.5 tiene sentido también hablar de aristas eficientes en $f(X)$.

Teorema 4.4. *Sean $f(x^i)$ y $f(x^j)$ dos puntos extremos adyacentes en el espacio objetivo. Entonces, partiendo de x^i se puede obtener x^j.*

Demostración. Hay que considerar dos casos:

(i) Si x^i y x^j son adyacentes en el espacio de decisiones, mediante la arista eficiente $\gamma(x^i, x^j)$.

(ii) Si no son adyacentes, mediante la secuencia de aristas eficientes $\gamma(x^i, x^{i+1}),...,\gamma(x^{j-1}, x^j)$. □

Dado un punto extremo eficiente en el espacio objetivo, el mayor número de puntos extremos eficientes adyacentes a este es dos. En ausencia de degeneración, dado un punto extremo eficiente en el espacio objetivo es posible encontrar un punto extremo eficiente adyacente a este, introduciendo en la correspondiente base del primero un conjunto de variables no básicas.

En caso de degeneración, es decir, si dado un punto extremo eficiente $x^i \in X$ existe más de una base que corresponde a ese punto, entonces también todas esas bases corresponden al mismo punto $f(x^i)$. En el proceso para encontrar un punto extremo eficiente en el espacio objetivo necesitamos, por tanto, calcular la base eficiente entre las bases degeneradas.

3.1 Algoritmo

Las ideas previas permiten el desarrollo de un algoritmo cuyas ideas esenciales aparecen en Sedeño-Noda y González-Martín [71]. El método propuesto utiliza los índices de árbol *Pred*, *Depth* y *Thread* para mejorar las operaciones de actualización en el proceso de pivoteo (ver por ejemplo, Ahuja et al. [4], Glover et al. [35] y capítulo 1). El proceso de pivoteo en nuestro método está constituido por una secuencia finita de pivoteos del método Simplex para Redes uniobjetivo. Como se indicó con anterioridad, los arcos candidatos para entrar en la base son aquellos que no cumplen las condiciones de optimalidad con respecto al segundo objetivo. Es decir, los arcos (i,j) no básicos tales que: $\bar{c}_{ij}^2 < 0 / (i,j) \in L$ ó $\bar{c}_{ij}^2 > 0 / (i,j) \in U$. De entre estos, son elegidos aquellos asociados a una

arista eficiente en el espacio objetivo. El algoritmo tiene el siguiente esquema:

```
procedure Computar_Nuevo_punto(x,B,L,U,π1,π2,Pred,Depth,Thread,
S);
  begin
  while S <>∅ do
    begin
    Sea (i,j) el primer_arco de S; S := S – (i,j);
    if (i,j) no cumple la condiciones de optimalidad con
    respecto al segundo objetivo then
      begin
      Realiza un pivoteo con el arco entrante (i,j);
      Actualiza x, B, L, U, π1, π2, Pred, Depth, Thread;
      end
    end
  end;

procedure Computar_Arcos_Entrantes(L, U, c, π1, π2, S, θt);
  begin
  c̄ij^k = cij^k – πi^k + πj^k, ∀(i,j) ∈ A, k = 1,2;
  S := ∅;
  Sea θt := min{ c̄ij^2 / c̄ij^1 : c̄ij^2 < 0  ∀(i,j) ∈ L, c̄ij^2 / c̄ij^1 : c̄ij^2 > 0  ∀(i,j) ∈ U };
  Sea S el conjunto de arcos donde se alcanza el anterior mínimo
  end;

Algoritmo EEO1;
  begin
  Resolver el problema P(λ) con λ=1 obteniendo x0, π1, π2 y la
  estructura de árbol generador (B,L,U);
  Sean Pred, Depth y Thread los índices del árbol;
  Almacenar x0 como punto extremo eficiente en el espacio
  objetivo;
  Hacer t = 1;
  Computar_Arcos_Entrantes(L, U, c, π1, π2, S, θt);
  while S <> ∅ do
    begin
    xt:= xt-1;
    Computar_Nuevo_punto(xt,B,L,U,π1,π2,Pred,Depth,Thread,S);
    if xt <> xt-1 then
      begin
      Almacena x t como nuevo punto extremo eficiente en el
      espacio objetivo;
      λt = |θt| / (1 + |θt|);
      t := t +1

      end;
    Computar_Arcos_Entrantes(L, U, c, π1, π2, S, θ t)
```

```
    end
  end.
```

El algoritmo parte del punto extremo eficiente en el espacio objetivo que resulta de optimizar sólo el primer objetivo. El procedimiento *Computar_Arcos_Entrantes* se encarga de calcular los arcos que violan las condiciones de optimalidad con respecto al segundo objetivo y que conforman la secuencia de pivoteos para alcanzar el punto extremo eficiente adyacente al anterior en el espacio objetivo. El procedimiento *Computar_Nuevo_punto* se encarga de llevar a cabo estos pivoteos actualizando la estructura de árbol generador, los índices del árbol y los potenciales con respecto a los dos objetivos. Si, en este proceso, el flujo x^{t-1} ha sido modificado, entonces hemos computado el punto x^t. Es decir, el punto extremo eficiente $f(x^t)$ adyacente a $f(x^{t-1})$ en el espacio objetivo. Este nuevo punto es óptimo para el problema $P(\lambda)$ con $\lambda^t \leq \lambda \leq \lambda^{t-1}$. Al finalizar el proceso, se obtiene el conjunto de todos los puntos extremos eficientes en el espacio objetivo.

Como hemos dicho previamente, nuestro método toma como base el procedimiento de Lee y Pulat [53], pero difiere de este en que los arcos introducidos en el pivoteo para computar un nuevo punto extremo en el espacio objetivo son todos los asociados a una arista eficiente en dicho espacio. Esto hace que sea menor el número de puntos extremos examinados y el número de veces que son examinadas las variables no básicas. Además, nuestro método es un Simplex para redes biobjetivo. En cambio Lee y Pulat modifican el método Out-of-Kilter, incluyendo en el proceso de etiquetado la construcción de una base para la obtención del camino a lo largo del cual se pueda enviar flujo. En nuestro caso, el Simplex nos suministra la base. Por todo esto, el esfuerzo computacional de nuestro algoritmo es sustancialmente menor. Además, sólo se calculan puntos extremos en el espacio objetivo. Estas dos últimas observaciones se harán patentes más adelante.

Ya hemos mencionado que, una vez que son obtenidos los puntos extremos eficientes en el espacio objetivo, es posible determinar todos los puntos eficientes en el espacio de decisiones. Para realizar esto, únicamente es necesario que el algoritmo EEO1 almacene el par (x, S) correspondiente a cada punto extremo eficiente en el espacio objetivo. Así, en una segunda fase, cada

punto extremo eficiente en el espacio de decisiones puede ser generado realizando un pivoteo con cada arco no básico en el conjunto S con respecto al flujo x, y así sucesivamente.

3.2 Un ejemplo

A continuación, consideramos el ejemplo introducido por Pulat et al. [62], dado en la Figura 4.1, donde $b_1 = 10$, $b_6 = -10$ y $b_i = 0$, para $i = 2,..,5$.

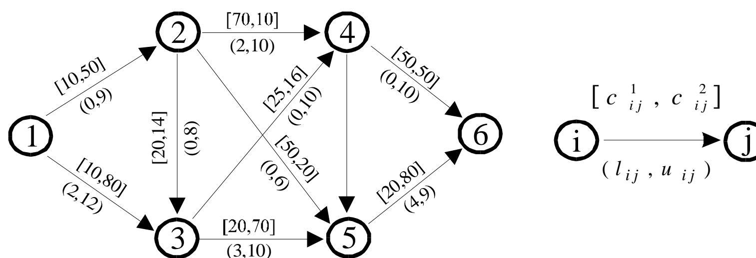


Figura 4.1. Red del ejemplo.

Este ejemplo tiene 8 puntos eficientes en el espacio de decisiones, de los cuales 4 corresponden a puntos extremos en el espacio objetivo (x^1, x^2, x^3 y x^8). Estos puntos se pueden ver en la Tabla 4.1. Los arcos básicos para cada solución eficiente son destacados en negrita y aquellas soluciones eficientes que son puntos extremos en el espacio objetivo aparecen subrayadas.

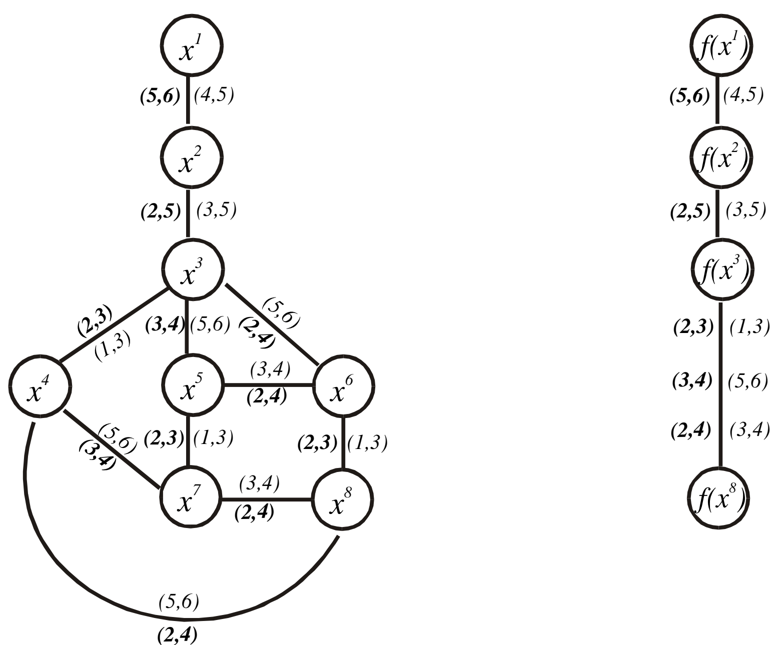
	x_{ij}										f_1	f_2
	(1,2)	(1,3)	(2,3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)	(4,6)	(5,6)		
$\underline{x^1}$	**2**	**8**	0	2	0	0	**8**	1	**1**	9	640	2170
$\underline{x^2}$	**2**	**8**	0	2	0	0	**8**	0	**2**	**8**	660	2060
$\underline{x^3}$	**7**	**3**	0	2	**5**	0	3	0	**2**	**8**	810	1660
x^4	**8**	2	**1**	2	**5**	0	3	0	**2**	**8**	830	1644
x^5	**3**	**7**	0	2	**1**	**4**	3	0	**6**	4	830	1644
x^6	**7**	**3**	0	**6**	**1**	0	3	0	**6**	4	1010	1500
x^7	**8**	2	**5**	2	**1**	**4**	3	0	**6**	4	930	1564
$\underline{x^8}$	**8**	2	**1**	**6**	**1**	0	3	0	**6**	4	1030	1484

Tabla 4.1. Soluciones eficientes en el espacio de decisiones y sus correspondientes valores objetivos.

A continuación, en la Figura 4.2 se muestra el grafo de adyacencias de puntos extremos eficientes en el espacio de decisiones y en el espacio objetivo. Para cada arista eficiente en el espacio de decisiones se muestra, en negrita, el arco entrante y, sin negrita, el arco saliente. Para cada arista eficiente en el espacio

objetivo se muestran el conjunto de arcos entrantes en negrita y sin negrita para los salientes.

En este ejemplo la solución óptima correspondiente para λ = 1 es x^1 (es óptimo para $0{,}8461538 \le \lambda \le 1$). El menor valor de θ se obtiene para el arco no básico (5,6) con θ = – 5,5. La entrada en la base de (5,6) trae consigo la salida de la misma de (4,5), obteniendo el punto extremo x^2, el cuál es óptimo para $0{,}7222222 \le \lambda \le 0{,}8461538$. En este momento, el valor de θ = – 2,666666 para la variable (2,5). La entrada en la base de (2,5) y la salida de (3,5) da como resultado el punto extremo x^3, el cuál es óptimo para $0{,}4444444 \le \lambda \le 0{,}7222222$. En x^3, las variables no básicas entrantes son {(2,3), (3,4), (2,4)} con θ = – 0,8. La entrada en la base de estas variables lleva consigo la salida de las variables {(1,3), (5,6), (3,4)}, obteniéndose el punto extremo x^8, el cuál es óptimo para $0 \le \lambda \le 0{,}4444444$. En este punto todas las variables no básicas cumplen las condiciones de optimalidad con respecto al segundo objetivo.



Grafo de adyacencia de soluciones eficientes en el espacio de decisiones

Grafo de adyacencia de soluciones eficientes en el espacio objetivo

Figura 4.2. Grafo de adyacencia de puntos extremos.

Si observamos el conjunto de arcos entrantes y el conjunto de arcos salientes en x^3, vemos que ambos conjuntos no son disjuntos. Dicho de otro modo, para alcanzar a x^8 desde x^3, es suficiente que los arcos que entren en la base sean *(2,3)* y *(2,4)* (ver Figura 4.2). En otras palabras, en el grafo de adyacencia de puntos extremos eficientes en el espacio de decisiones existen caminos alternativos

conectando x^3 y x^8. El camino seleccionado por el algoritmo dependerá del orden en el que los arcos candidatos sean elegidos.

El experimento computacional realizado con este algoritmo mostrará que el tiempo de CPU consumido es relativamente pequeño. Dicho experimento, será expuesto una vez que introduzcamos el segundo algoritmo para resolver el problema FRB.

4. Estudio para la distancia del máximo ponderada

Para el caso de la métrica del máximo ponderado o métrica de Tchebycheff la distancia viene dada por la siguiente expresión:

$$d_{\pi}(f^*, f(x)) = \max_{k=1,2} \ \pi_k \left(f_k(x) - f_k^*\right)$$

A diferencia del caso lineal, donde los pesos son obtenidos vía programación paramétrica, los pesos en la métrica del máximo ponderado deben ser dados por un decisor (o se puede simular la presencia de un decisor). Evidentemente, el decisor puede no entender de pesos, pero seguramente entenderá de los niveles alcanzados por los objetivos. Además, puede desear alcanzar unos niveles, para cada uno de estos objetivos, que le resulten satisfactorios. En este sentido, los pesos π se construyen de la siguiente manera (González-Martín [41]): Para cada objetivo el decisor introduce un nivel de aspiración, $z_k > f_k^*, k = 1,2$, y entonces:

$$\pi_k = \frac{1/(z_k - f_k^*)}{\sum_{i=1,2} 1/(z_i - f_i^*)}$$

El problema P_π, con la métrica d_π y los pesos π, tiene como solución, el punto eficiente más cercano, de acuerdo con la dirección de búsqueda definida por el punto ideal y el vector de nivel de aspiración z.

Teorema 4.5 (González-Martín [41]).

(i) *La solución del problema P_π es un punto eficiente del problema FRB que se corresponde con la intersección entre el conjunto de puntos eficientes E[f(X)] y la recta definida por f^* y el vector de nivel de aspiración z.*

(ii) *Existe una correspondencia uno a uno entre soluciones eficientes del problema FRB y pesos π generados de la manera anterior.*

4.1 Resolución del problema P_π

En nuestro caso, el problema P_π puede ser expresado de la manera siguiente:

$$\overline{y} = \min \; y \qquad (4.4a)$$

$$s.t: \pi_k \left(\sum_{i \in V} \sum_{j \in Suc(i)} c_{ij}^k x_{ij} - f_k^* \right) \le y, k = 1,2 \qquad (4.4b)$$

$$\sum_{j \in \mathrm{Suc(i)}} x_{ij} - \sum_{j \in Pred(i)} x_{ji} = b_i, \forall \mathrm{i} \in \mathrm{V} \qquad (4.4c)$$

$$l_{ij} \le x_{ij} \le u_{ij}, \forall (i,j) \in A \qquad (4.4d)$$

Las restricciones (4.4c) y (4.4d) son las clásicas restricciones del problema de flujo de coste mínimo en una red. La restricción (4.4b) está relacionada con el nivel de aspiración para cada objetivo que es incorporado al problema mediante los pesos π. El objetivo (4.4a) contiene la minimización de la mayor de las desviaciones alcanzadas por los diferentes objetivos con respecto al punto ideal.

La restricción (4.4b) rompe la propiedad de unimodularidad del problema de flujo de coste mínimo. Sin embargo, esta dificultad puede ser superada si la mencionada restricción es incorporada en la función objetivo mediante Relajación Lagrangiana:

$$\min \; y + \sum_{k=1}^{2} w_k \left(\pi_k \left(\sum_{i \in V} \sum_{j \in Suc(i)} c_{ij}^k x_{ij} - f_k^* \right) - y \right) \qquad (4.5a)$$

$$\mathrm{s.t}: \sum_{j \in \mathrm{Suc(i)}} x_{ij} - \sum_{j \in Pred(i)} x_{ji} = b_i, \forall \mathrm{i} \in \mathrm{V} \qquad (4.5b)$$

$$l_{ij} \le x_{ij} \le u_{ij}, \forall (i,j) \in \mathrm{A} \qquad (4.5c)$$

$$w_k \ge 0, k = 1,2 \qquad (4.5d)$$

donde w es el vector de multiplicadores de Lagrange.

Ordenando los términos de (4.5a) se obtiene la siguiente función objetivo:

$$y(1-\sum_{k=1}^{2} w_k) - \sum_{k=1}^{2} w_k \pi_k f_k^* + \sum_{k=1}^{2} w_k \pi_k \sum_{i\in V} \sum_{j\in Suc(i)} c_{ij}^k x_{ij}$$

Se puede notar que los dos primeros términos de la expresión anterior no dependen de x. En este caso, fijado w, el problema de Flujo de Coste Mínimo uniobjetivo a resolver es:

$$L_1(w) = \min\left(\sum_{i\in V} \sum_{j\in Suc(i)} \sum_{k=1}^{2} w_k \pi_k c_{ij}^k x_{ij} \,/\, s.a.: (4.5b),\ (4.5c) \right) \quad (4.6)$$

Definimos

$$L(w) = L_1(w) - \sum_{k=1}^{2} w_k \pi_k f_k^* + y(1-\sum_{k=1}^{2} w_k)$$

y resolvemos el problema de Multiplicadores de Lagrange

$$\overline{L} = \max_{w\geq 0} L(w) \quad (\mathrm{ML})$$

Si $\overline{y}$ es la solución óptima del problema P_π, para cualquier elección del vector de multiplicadores de Lagrange w, y cualquier solución factible x del problema P_π, entonces $L(w) \leq \overline{L} \leq \overline{y} \leq y$. Además, si w es un vector de multiplicadores de Lagrange y x es una solución factible del problema P_π satisfaciendo $L(w)=y$, entonces w es una solución óptima del problema ML y x es una solución óptima para el problema P_π.

Teorema 4.6. *El valor óptimo del problema ML coincide con el valor óptimo del problema P_π, es decir, $\overline{L} = \overline{y}$ (ver teorema 4.5).*

Por tanto, la resolución de (4.4) puede hacerse resolviendo adecuadamente los problemas (4.6) y ML. Para ello, damos un valor inicial al vector de multiplicadores de Lagrange w (por ejemplo, todas las componentes iguales a 1) y, en cada iteración, modificamos los valores de w de acuerdo con el método del

subgradiente, hasta que se calcula $\bar{L}$. En este proceso usamos el método Simplex para redes para resolver los problemas del tipo (4.6). Al finalizar, obtendremos el punto extremo eficiente más cercano al punto eficiente solución de P_π.

Sea $\bar{f} \in f(X)$ una solución eficiente de P_π y $\bar{f}_{ex} \in f(X)$ el punto extremo eficiente más cercano a $\bar{f}$ de acuerdo con la distancia del máximo ponderado d_π (ver Figura 4.3).

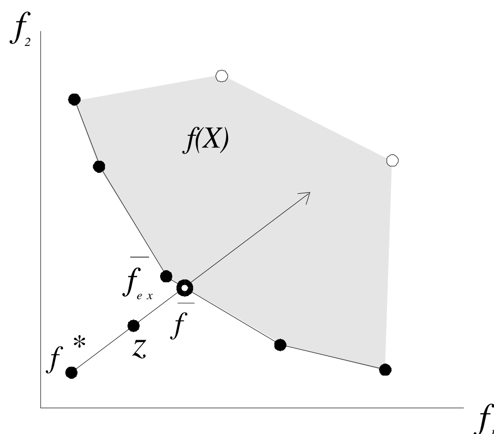


Figura 4.3. Espacio objetivo.

Un problema que puede aparecer es que el método Simplex para redes, cuando se aplica para resolver (4.6), es incapaz de calcular $\bar{f}$, si este punto no es un punto extremo de *f(X)*. Para impedir que se de esta situación, se obtienen cotas superiores de $\bar{y}$. Para esto es necesario identificar algunos puntos extremos del poliedro *f(X)*, es decir, determinar la frontera inferior definida por los puntos extremos más cercanos a la solución eficiente. Sea $\tilde{f}$ el punto intersección entre la recta $t = f^* + \lambda(z - f^*)$ y el segmento H_p definido por los dos puntos extremos eficientes conocidos que son más cercanos al punto solución. Podemos asignar a y el valor de d_π que se obtiene con el punto $\tilde{f} = f^* + \lambda(z - f^*)$. De esta manera, al finalizar nuestro método $\tilde{f}$ coincidirá con $\bar{f}$.

En el caso biobjetivo, la solución eficiente $\bar{f}$ se obtiene siempre resolviendo el problema P_π. Esto es debido a que al finalizar la resolución de P_π, el segmento H_p coincide con una arista eficiente del poliedro *f(X)*.

Si reordenamos los términos de la función objetivo del problema de relajación Lagrangiana (4.5) según los valores de $\tilde{f}$, tenemos:

$$\tilde{y} + \sum_{k=1}^{2} w_k \left(\pi_k \left(\sum_{i \in V} \sum_{j \in Suc(i)} c_{ij}^k x_{ij} - f_k^* \right) - \tilde{y} \right) = \tilde{y} - \sum_{k=1}^{2} w_k \pi_k \tilde{f}_k + \sum_{k=1}^{2} w_k \pi_k \sum_{i \in V} \sum_{j \in Suc(i)} c_{ij}^k x_{ij} \quad (4.7)$$

donde $\tilde{y} = \pi_k(\tilde{f}_k - f_k^*),\ k = 1,2$ (teorema 4.5).

Si denotamos por $\tilde{c}_{ij} = \sum_{k=1}^{2} w_k \pi_k c_{ij}^k$, la expresión (4.6) coincide con $\tilde{y} - \sum_{k=1}^{2} w_k \pi_k \tilde{f}_k + \sum_{i \in V} \sum_{j \in Suc(i)} \tilde{c}_{ij} x_{ij}$, donde los dos primeros términos son conocidos.

La técnica del subgradiente nos permite obtener $\tilde{f}$. Para ello almacenamos en una lista los dos puntos extremos más cercanos en orden no decreciente según los valores de $d_\pi = \max_{k=1,2} \left(\pi_k \left(f_k - f_k^* \right) \right)$. A continuación procedemos a detallar el algoritmo para resolver P_π.

4.2 Algoritmo para el problema P_π

Al principio del procedimiento el segmento H_p esta definido por los puntos extremos asociados con los niveles ideales de cada objetivo. Sea $\bar{f}_{ex}$ el mejor punto extremo eficiente conocido. Al finalizar el algoritmo, este punto almacenará el punto solución. $\tilde{f}_{ex}$ es el nuevo punto extremo que es calculado en cada iteración. El algoritmo tiene el siguiente esquema:

```
Algoritmo Pπ;
  begin
  Hp:= segmento definido por los puntos extremos asociados con
  cada nivel ideal de los objetivos;
  f̃ := punto resultante de la intersección entre Hp y la
  dirección de búsqueda;
  ỹ = πk(f̃k − fk*),k = 1,2;
  f̄ex:= Primer punto de la lista Hp;
  q := 1; {iteración}; λ := 2.0; {factor de escala};
```

```
w_k^q := 1; k = 1,2 {inicialización de multiplicadores de Lagrange}
while (ỹ <>L(w)) do
  begin
  Sea f̃_ex la solución de L1(w) obtenida con el método Simplex
  para redes;
  if f̃_ex es más cercano que cualquier otro punto de Hp then
    begin
    Intercambia f̃_ex con el punto extremo de Hp de tal manera
    que la intersección con la dirección de búsqueda sea
    distinta del vacío;
    f̃ := punto resultante de la intersección entre Hp y la
    dirección de búsqueda;
    ỹ = π_k(f̃_k − f_k^*), k = 1,2;
    f̄_ex := Primer punto de la lista Hp
    end;
  if L(w) no ha mejorado en r iteraciones then λ := λ/2;
  φ := λ(ỹ − L(w)) / ||(π_k(f̃_ext k − f̃_k), k = 1,2||^2 ; {siguiente tamaño del
  paso}
  w_k^{q+1} := { w_k^q + θ(π_k(f̃_ext k − f̃_k))   Si w_k^q + θ(π_k(f̃_ext k − f̃_k)) > 0   k = 1,2;
              { w_k^q                            en otro caso
  {siguiente w}
  q := q+1 {iteración siguiente}
  end
end.
```

El algoritmo comienza con w=1 y λ=2.0. En cada iteración se resuelve $L(w)$, obteniendo un punto extremo almacenado en $\tilde{f}_{ex}$. Si el valor de d_π asociado con este punto extremo es inferior que el de cualquier otro en H_p, este punto es reemplazado por el punto extremo más lejano del segmento. El nuevo segmento debe tener una intersección no nula con la recta definida por f^* y z. Esta operación dará un nuevo punto $\tilde{f}$ y una nueva cota superior $\tilde{y}$ que mejora su valor anterior. Este valor es utilizado para calcular el tamaño del paso y para calcular los multiplicadores de Lagrange de la siguiente iteración. Si el valor de $L(w)$ no ha aumentado con respecto a los valores obtenidos en las r iteraciones previas, (en nuestra implementación r=3), el factor de escala λ toma el valor λ/2. El tamaño del paso, θ, se calcula de acuerdo con el método de Newton para resolver sistemas de ecuaciones no lineales (ver Ahuja et al. [4]). Como el valor óptimo $\bar{y}$ debe coincidir con $\bar{L}$, podemos utilizar el criterio de parada: $\tilde{y}$ =$L(w)$.

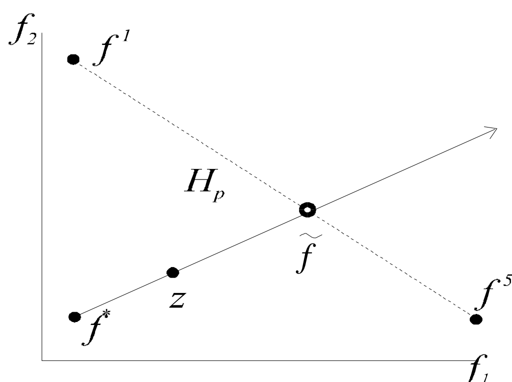
4.3 Un ejemplo

Para ilustrar las ideas generales de este proceso, usaremos un ejemplo presentado por Aneja y Nair [9] y usado por Ringuest y Rinks [63]. Este ejemplo corresponde a un problema de transporte biobjetivo que puede ser fácilmente convertido en un problema de flujo de coste mínimo. Los datos del problema son los siguientes:

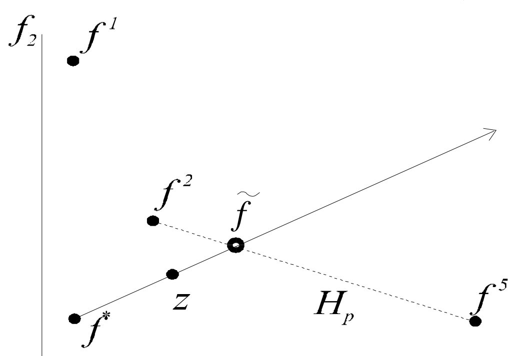
Puntos		***Puntos destinos***				
		1	2	3	4	a_i
	1	(1,4)	(2,4)	(7,3)	(7,4)	8
	2	(1,5)	(9,8)	(3,9)	(4,10)	19
	3	(8,6)	(9,2)	(4,5)	(6,1)	17
	b_j	11	3	14	16	

Donde el valor que aparece en cada celda de la tabla es $\left(c_{ij}^1, c_{ij}^2\right)$.

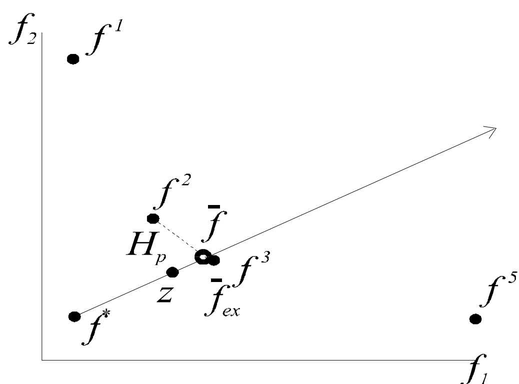
Este problema tiene como soluciones los puntos extremos eficientes f^1=(143,265), f^2=(156,200), f^3=(176,175), f^4=(186,171), f^5=(208,167). El punto ideal es f^*=(143,167). Bajo la suposición de que z =(160,180), el correspondiente vector de pesos es π =(0.433333, 0.566667). Por lo tanto, debemos obtener el punto extremo más cercano a la solución del problema P_π. El proceso se puede observar en las Figuras siguientes:



La primera vez el segmento H_p está definido por los puntos extremos f^1, f^5. La cota superior es $\tilde{y} = \pi_k(\tilde{f}_k - f_k^*)$, que es la misma para todo valor de k, ya que $\tilde{f}$ está sobre la dirección de búsqueda. En este caso, se tiene que $\tilde{f}$ = (186.12624, 199.9788) e $\tilde{y}$ = 18.6880. El vector de multiplicadores inicial es w=(1,1).



A continuación se resuelve el problema $L_1(w)$ obteniendo el punto extremo f^2. Ahora el segmento H_p esta definido por f^2, f^5 ($d_\pi(f^*,f^2) = 18{,}70$ < $d_\pi(f^*,f^1) = 55{,}33$). Se tiene que: $\tilde{f}$ =(172.4785, 189.5424) e $\tilde{y}$ =12.7740. El vector de multiplicadores es actualizado y, como $\tilde{y} \neq L(w)$, y el algoritmo



continúa.
Otra vez se resuelve el problema $L_1(w)$, obteniendo es este caso el punto extremo f^3. En este momento H_p esta definido por f^2, f^3 ($d_\pi(f^*, f^3) = 14{,}29$ < $d_\pi(f^*, f^5) = 28{,}16$). Los valores obtenidos son $\tilde{f}$ =(167.4452, 185.6934) e $\tilde{y}$ =10.5929. Una vez obtenido el punto extremo f^3, los puntos extremos calculados en subsiguientes iteraciones serán f^2 y f^3 repetidamente. El algoritmo termina cuando $L(w)$ coincide con $\bar{y}$.

Figura 4.4. Traza del ejemplo.

Al finalizar el algoritmo, $\bar{f} = \tilde{f}$ es la solución eficiente del problema y $\bar{f}_{ex} = f^3$ es el punto extremo más cercano al punto solución en la dirección dada. El punto extremo f^4 no es calculado, debido a que no es necesario para obtener la solución del problema P_π.

En la resolución del problema P_π, los puntos extremos calculados pertenecen al cono definido por los puntos extremos en el segmento H_p y el punto ideal. En cada iteración únicamente es computado un punto extremo perteneciente a este cono. Si este punto cumple las condiciones dadas en el método anterior, el cono será más pequeño y se corresponde con puntos de R^2 con valores menores de y. En la Figura 4.5 se muestran los conos sucesivos para el ejemplo anterior.

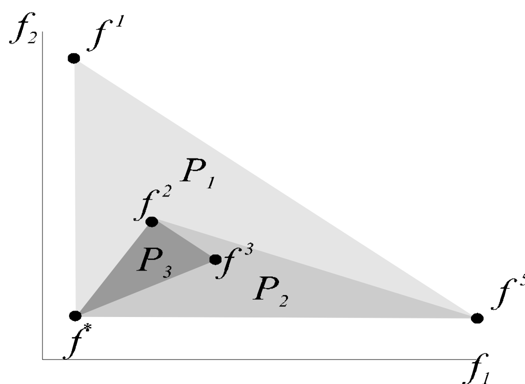


Figura 4.5. Los sucesivos conos.

Una vez resuelto el problema P_π, podemos idear un método para obtener todos los puntos extremos eficientes en el espacio

objetivo del problema FRB. Además, podemos diseñar un método para obtener los puntos extremos pertenecientes a una región especial de interés para el decisor en el espacio objetivo.

4.4 Algoritmo para el problema FRB

El método para calcular todos los puntos extremos eficientes está basado en las ideas previas. Dada una dirección de búsqueda, especificada por el vector de niveles de aspiración z y el punto ideal, podemos determinar el punto extremo eficiente más cercano, de acuerdo con la dirección dada, usando el anterior algoritmo.

En el transcurso de la resolución del correspondiente problema P_π, podemos aprovechar el hecho de que otros puntos extremos eficientes son calculados antes de obtener la solución del problema. Podemos usar esta ventaja para diseñar un método que compute todos los puntos extremos eficientes.

Al principio únicamente conocemos los puntos extremos eficientes que minimizan cada objetivo separadamente y, por lo tanto, el segmento H_p está definido por estos puntos. A continuación, se construye una dirección adecuada y el correspondiente problema P_π se resuelve de la manera siguiente:

- Si es determinado un punto extremo no calculado con anterioridad (aunque no sea el óptimo para el problema P_π), son construidos dos segmentos. Estos nuevos segmentos resultan de la sustitución gradual del punto extremo calculado por cada uno de los puntos extremos del segmento actual.

- Si no es determinado un nuevo punto extremo, significa que, para el cono definido por los puntos extremos en el segmento y el punto local ideal, no existe otro punto extremo. Por lo tanto, este segmento no es considerado en el futuro.

El método que proponemos usa una cola, Q, que almacena cada uno de los segmentos que determinan los conos a examinar. Para cada uno de ellos se construye un vector arbitrario de niveles de

aspiración z de tal manera que pertenezca al cono examinado; esta operación determina un vector de pesos π. A continuación el correspondiente problema P_π se resuelve. Una vez que el actual segmento es examinado, el método continúa con el siguiente segmento de la cola. El algoritmo termina cuando la cola está vacía. El método propuesto tiene el siguiente esquema:

```
Algoritmo EEO2;
  begin
  Hp-actual:= segmento definido por los puntos extremos obtenidos
  en la minimización de cada objetivo;
  Pex := los puntos de Hp-actual;
  Inicializar _ cola(Q);
  poner_en_cola(Q,Hp-actual);
  while (Q <> ∅) do
    begin
    Hp-actual := Primer_elemento_de(Q);
    f*L := punto ideal local asociado con Hp-actual;
    Nuevo_punto_extremo := false;
    π := Obtener unos pesos adecuados;
    Resolver Pπ (Hp-actual,f*L , π, Nuevo_punto_extremo,fext );
    if Nuevo_punto_extremo then
      begin
      Pex := Pex +fext ;
      Añadir a la cola Q los dos segmentos que resultan del
      intercambio de fext con cada uno de los puntos en Hp-actual
      end
    end
  end.
```

Los puntos extremos calculados por el algoritmo son almacenados en Pex. $H_{p\text{-}actual}$ es el segmento de búsqueda que está siendo actualmente examinado por el algoritmo y que, junto con el punto ideal local f_L^*, define el cono de búsqueda. Dada una dirección y el cono de búsqueda, el correspondiente problema P_π será entonces resuelto. Si, en la resolución de este problema se obtiene un nuevo punto extremo (en el algoritmo, f_{ext}), el proceso termina. En este caso, se introduce f_{ext} en Pex, y los nuevos segmentos creados son añadidos a la cola Q.

El vector z en cada cono examinado puede ser elegido de la manera siguiente. Sean f^r y f^s los puntos extremos que junto con el punto ideal local f_L^* definen el cono examinado. Entonces, hacemos cada z_k igual a $z_k = \left|f_k^r - f_k^s\right| / 2 + f_{L_K}^*, k = 1,2.$ No obstante, no hay problema

si el vector z está dentro o fuera de la región eficiente debido al teorema 4.5 (z determina un vector de pesos π).

La Figura 4.6 ilustra la idea del método para el ejemplo introducido previamente. En la Figura 4.6a), el primer segmento está definido por f^1 y f^5, debido a que estos son los puntos que minimizan cada objetivo separadamente. En este caso, el punto ideal local coincide con el punto ideal f^*. Supongamos que la dirección de búsqueda es la indicada en la figura, y que, resolviendo P_π, obtenemos f^2 (aunque el punto extremo solución es f^3).

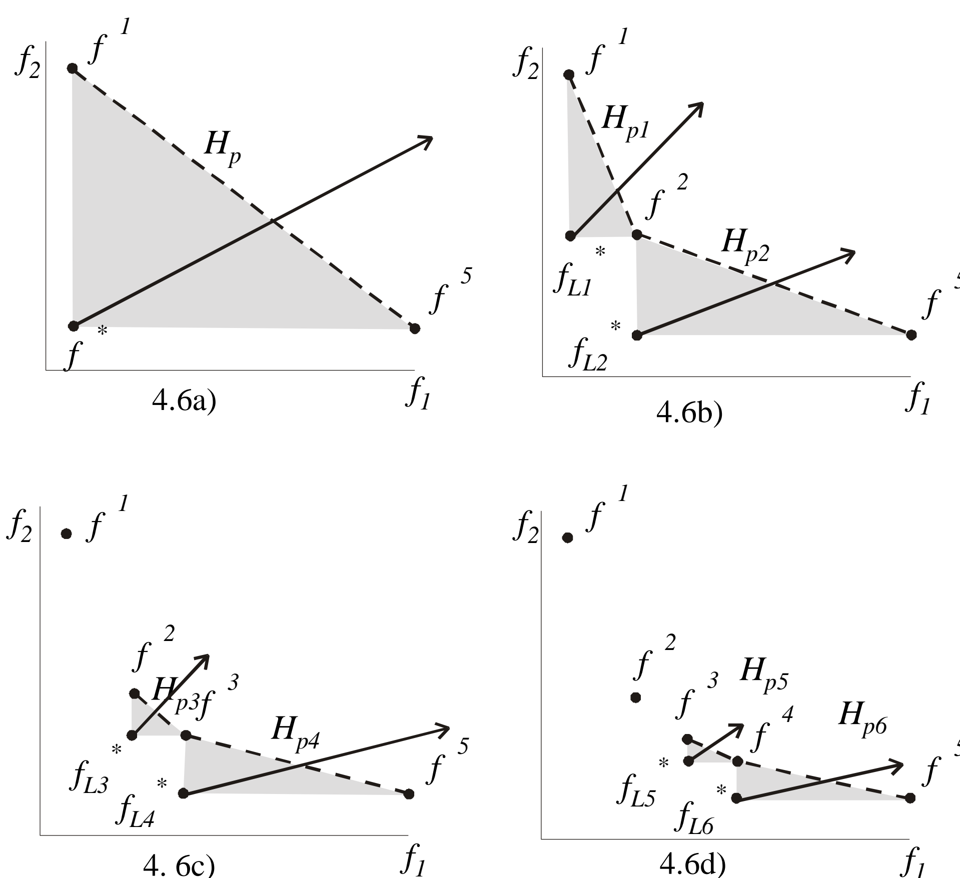


Figura 4.6.

En la Figura 4.6b), una vez determinado f^2, se muestran los segmentos generados. Ahora nos enfrentamos a dos problemas similares al original y, por lo tanto, damos una dirección de búsqueda para cada cono y resolvemos el problema P_π asociado con cada uno de ellos. En la Figura 4.6c), se pueden observar los dos nuevos segmentos construidos (H_{p3} y H_{p4}) cuando el punto f^3 ha sido encontrado. Podemos observar que en el cono definido por H_{p1}, no ha sido calculado ningún nuevo punto extremo. Por lo tanto, el

método no examina más está región. En la Figura 4.6d), se muestran los dos nuevos segmentos, H_{p5} y H_{p6}, cuando se calcula f^4. Ningún punto extremo adicional es encontrado en H_{p3} y, por lo tanto, no se generan nuevos segmentos. Finalmente, no hay nuevos puntos extremos en los conos H_{p5} y H_{p6} y, así, el método termina identificando todos los puntos extremos eficientes.

Teorema 4.7. (*Validación del algoritmo*). *El algoritmo anterior determina todos los puntos extremos eficientes que pertenecen a los conos de búsqueda.*

Demostración. Supongamos que el segmento H_p y el punto ideal local f_L^* definen uno de los conos de búsqueda y que únicamente el punto extremo f^q pertenece a este cono. Como f^q es un punto extremo, no pertenecerá al segmento H_p.

Sea z un vector arbitrario de niveles de satisfacción perteneciente a este cono. Los vectores z y f_L^* determinan un vector de pesos π. Sea $\tilde{f}$ el punto resultante de la intersección de la dirección con el segmento H_p, y sea $\tilde{y}$ su correspondiente valor objetivo para el problema P_π. Sea $\bar{f}$ la solución eficiente e $\bar{y}$ el valor óptimo del problema P_π. Claramente, $\bar{f} < \tilde{f}$ y, por lo tanto, $\bar{y} < \tilde{y}$, debido a que el segmento definido por la sustitución de f^q por un punto de H_p es más cercano al punto ideal local. Por esta razón, en la resolución del problema P_π, el punto extremo f^q es calculado dado que es más cercano que al menos uno de los puntos de H_p al punto eficiente solución.

4.5 Complejidad teórica del algoritmo

El método presentado depende del número de puntos extremos contenidos en el conjunto solución del problema FRB. En otras palabras, la complejidad del caso peor del algoritmo viene dada por el número de puntos extremos eficientes multiplicado por el esfuerzo computacional requerido en la resolución del problema P_π. Esto es debido a que cada vez que un nuevo punto extremo es encontrado, el algoritmo no persiste en examinar el

correspondiente cono de búsqueda. Para cada nuevo punto extremo se tienen en cuenta dos nuevos conos de búsqueda que requieren la resolución del correspondiente problema P_π. Así, la complejidad del caso peor del método es $O(|E_{ex}[f(X)]| \cdot S(P_\pi))$, donde $S(P_\pi)$ es la complejidad del caso peor en la resolución del problema P_π. En la resolución del problema P_π el método realiza, a lo sumo, 15 iteraciones. Debido a esto, $S(P_\pi)$ coincide con la complejidad del método Simplex para redes. Por lo tanto, el comportamiento empírico del algoritmo puede ser particularmente bueno para los casos del problema FRB con un pequeño número de puntos extremos eficientes. Esto se pondrá de manifiesto en los resultados computacionales.

4.6 Un ejemplo

Consideremos el ejemplo introducido por Pulat et al. [62] dado en la Figura 4.1. Dicho ejemplo tenía los siguientes cuatro puntos extremos en el espacio objetivo: f^1=(640,2170), f^2=(660,2060), f^3=(810,1660) y f^4=(1030,1484) (debemos recordar que en el espacio de decisiones hay 8 puntos extremos eficientes). El algoritmo EEO2 los identifica partiendo del conocimiento de los puntos extremos f^1 y f^4.

En la Tabla 4.2 *iter* representa la iteración, $H_{p\text{-}actual}$ es el segmento actual de búsqueda definido por los puntos extremos que están entre llaves, π almacena los pesos arbitrarios que definen la dirección de búsqueda. Q es la cola que almacena todos los segmentos que no han sido examinados, incluyendo el actual. f_{ext} es el nuevo punto extremo y *Pex* es la lista que almacena los puntos extremos.

Iter	$H_{p\text{-}actual}$	π	Q	f_{ext}	*Pex*
1	$\{f^1,f^4\}$	(0.637546,0.362454)	$\{\{f^1,f^4\}\}$	f^3	$\{f^1,f^4,f^3\}$
2	$\{f^3,f^4\}$	(0.239130,0.760870)	$\{\{f^3,f^4\},\{f^1,f^3\}\}$		$\{f^1,f^4,f^3\}$
3	$\{f^1,f^3\}$	(0.750000,0.250000)	$\{\{f^1,f^3\}\}$	f^2	$\{f^1,f^4,f^3,f^2\}$
4	$\{f^1,f^2\}$	(0.984399,0.015601)	$\{\{f^1,f^2\},\{f^2,f^3\}\}$		$\{f^1,f^4,f^3,f^2\}$
5	$\{f^2,f^3\}$	(0.833702,0.166298)	$\{\{f^2,f^3\}\}$		$\{f^1,f^4,f^3,f^2\}$

Tabla 4.2. Puntos extremos eficientes en el espacio objetivo

En la Tabla 4.2 se observa que en las iteraciones 2, 4, y 5, no se encuentra ningún punto extremo, debido a que en el correspondiente cono de búsqueda $H_{p\text{-}actual}$ no hay nuevos puntos extremos. En la iteración 3 todos los puntos extremos eficientes son conocidos.

5. Resultados computacionales de los algoritmos EEO1 y EEO2.

En esta sección, haremos comentarios sobre los códigos implementados, daremos los resultados computacionales y realizaremos la comparación de nuestros dos métodos con el algoritmo de Lee y Pulat.

Ambos algoritmos comienzan obteniendo el punto extremo eficiente que minimiza únicamente el primer objetivo. Para ello, hemos implementado un método Simplex para redes uniobjetivo en el que la prevención de ciclado y el fenómeno "stalling" han sido considerados. Nuestro método usa la regla de prevención de ciclado de Cunningham [21] y la regla LRB referida en Goldfarb y Hao [39] para anti-stalling.

Hemos elegido el método Simplex para redes por dos razones. La primera es que éste resuelve cualquier problema de flujos mientras que, por ejemplo, el método Out-of-Kilter resuelve únicamente el problema de circulación. La segunda razón es que existen estudios computacionales que muestran que el método Simplex para redes es substancialmente más rápido que los métodos Primal-Dual y Out-of-Kilter (ver Glover et al. [36]).

EEO1 y EEO2 son códigos PASCAL que han sido ejecutados en una estación de trabajo HP9000/712 a 60 MHZ. Hemos implementado también el método de Lee y Pulat, denotado por L&P. Hemos elegido este método para comparar con los que nosotros introducimos debido a que también fue diseñado para calcular todos los puntos extremos eficientes en el espacio objetivo. En Lee y Pulat [53] se afirma lo siguiente: "Si x^0 es un punto extremo eficiente en el espacio de decisiones entonces su imagen dada por $f(x^0)$ es un punto extremo en el espacio objetivo". Sin embargo, esto no es cierto siempre. Por ejemplo, en la red de la Figura 4.1, L&P

encuentra, además de los 4 puntos extremos que calculan EEO1 y EEO2, el punto (830,1644). Este punto corresponde a un extremo eficiente en el espacio de decisiones, pero no es extremo en el espacio objetivo. Por esta razón, cuando comparemos ambos métodos, EEO1 y EEO2 hallarán un menor número de puntos solución que L&P, ya que EEO1 y EEO2 únicamente calculan puntos extremos en el espacio objetivo.

Los problema test fueron generados usando NETGEN [52]. Debido a que el método L&P únicamente resuelve el problema de circulación, el número de nodos oferta/demanda es 1. El valor de los costes de la segunda función objetivo fueron generados de manera uniforme dentro del intervalo [-1000,1000]. Esos valores son enteros. En la Tabla 4.3 se muestran los niveles de número de nodos (n), número de arcos (m) el ratio m/n y la capacidad máxima de los arcos (U):

n	m	m/n	U
25	100, 175, 250	4, 7, 10	10, 1000, 100000
30	120, 210, 300	4, 7, 10	10, 1000, 100000
35	140, 245, 350	4, 7, 10	10, 1000, 100000
40	160, 280, 400	4, 7, 10	10, 1000, 100000

Tabla 4.3. Número de nodos, de arcos, ratio m/n y U.

Mediante las combinaciones de nodos (n), arcos (m) y capacidad máxima (U), obtenemos 36 parámetros de red. Para cada una de las combinaciones anteriores, hemos generado 5 replicas mediante NETGEN y las siguientes semillas: 12345678, 36581249, 23456183, 46545174, 35826749. Por lo tanto, el número de casos particulares de estudio fueron 180.

En la Tabla 4.4 se muestran los resultados computacionales de la comparación de EEO1, EEO2 y L&P. En esta tabla aparecen el número de nodos, de arcos, la capacidad máxima del grafo, la media y la desviación estándar del tiempo de CPU en segundos empleado por cada algoritmo. En esta tabla se puede ver que EEO1 es siempre mucho más rápido que EEO2 y L&P. En media, EEO1 es del orden de 50 veces más rápido que L&P. EEO2 y L&P emplean un tiempo de CPU similar, aunque EEO2 es más rápido que L&P para redes no densas.

En la Figura 4.7, mostramos el comportamiento de los algoritmos EEO2 y L&P con respecto al producto del número de

nodos y del número de arcos. Se puede observar que para grafos no densos, EEO2 es más rápido que L&P. Podemos destacar que en la práctica las redes suelen ser poco densas.

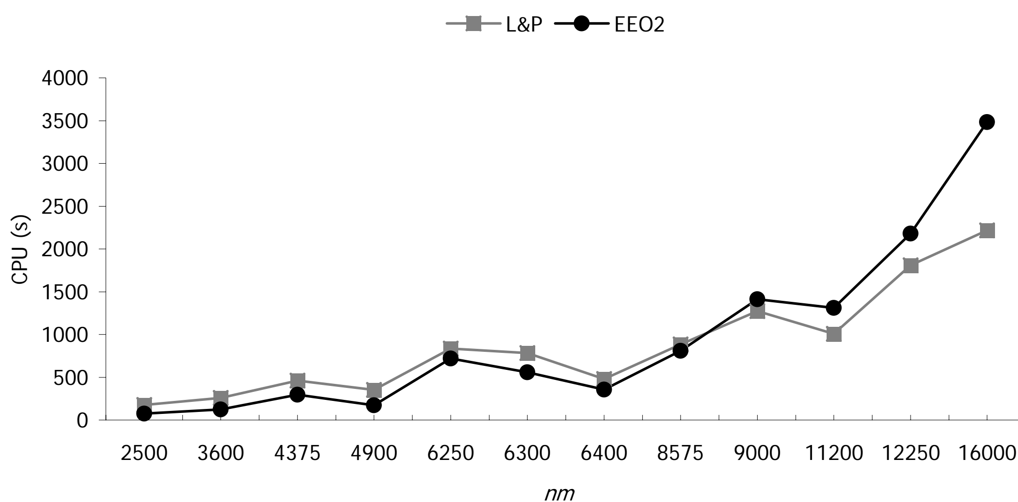


Figura 4.7. Tiempo de CPU con frente a número de nodos y arcos.

n	m	U	L&P CPU (s)		EEO1 CPU (S)		EEO2 CPU (s)		L&P	EEO1	EEO2
			Media	Desviación	Media	Desviación	Media	Desviación	Media de puntos extremos eficientes		
25	100	10	146,71	12,11	5,63	1,36	54,58	10,97	53,40	53,40	53,40
25	100	1000	189,09	16,89	6,78	1,09	94,36	24,80	71,60	71,60	71,60
25	100	1E+05	188,42	25,55	6,72	1,26	79,12	17,17	66,40	66,40	66,40
25	175	10	388,63	40,75	11,36	2,84	202,15	30,34	103,80	102,60	102,60
25	175	1000	496,59	61,10	12,73	3,33	333,13	84,10	135,00	134,80	134,80
25	175	1E+05	500,77	50,75	13,40	2,91	352,96	74,07	141,60	141,60	141,60
25	250	10	667,11	103,12	16,89	3,58	451,29	82,37	145,80	144,80	144,80
25	250	1000	899,99	132,99	18,66	5,25	889,37	207,40	214,60	214,20	214,20
25	250	1E+05	941,12	65,48	16,00	2,82	821,35	145,01	217,20	212,20	212,20
30	120	10	215,52	16,63	6,15	1,11	101,06	22,04	60,40	59,80	59,80
30	120	1000	293,36	20,44	6,37	1,39	139,90	35,16	68,40	67,80	67,80
30	120	1E+05	266,54	28,16	6,66	1,63	133,65	20,83	71,60	70,80	70,80
30	210	10	649,65	88,19	15,92	0,91	469,57	73,65	133,40	133,40	133,40
30	210	1000	878,17	73,86	16,45	3,28	572,69	113,39	157,80	154,00	154,00
30	210	1E+05	822,57	111,65	17,28	5,05	627,54	180,87	168,80	168,40	168,40
30	300	10	1003,17	111,46	23,97	4,66	1047,24	217,18	193,80	193,40	193,40
30	300	1000	1396,16	257,00	24,89	2,98	1527,45	175,41	252,40	252,20	252,20
30	300	1E+05	1421,59	283,84	25,94	5,66	1657,60	308,52	268,00	261,40	261,40
35	140	10	302,83	38,66	8,00	2,88	125,76	28,16	61,80	61,40	61,40
35	140	1000	364,55	45,13	8,81	4,10	193,20	72,64	87,40	87,20	87,20
35	140	1E+05	389,44	54,41	8,91	1,21	194,40	31,18	85,40	84,80	84,80
35	245	10	680,87	178,53	16,28	3,13	552,33	114,87	127,00	126,40	126,40
35	245	1000	938,01	359,11	17,58	5,25	881,32	250,52	176,00	176,00	176,00
35	245	1E+05	1036,92	179,48	17,58	5,33	988,81	283,45	181,20	181,20	181,20
35	350	10	1407,68	256,98	23,67	2,21	1441,51	230,83	197,60	197,60	197,60
35	350	1000	2007,48	352,58	29,16	2,13	2610,08	390,30	300,40	299,60	299,60
35	350	1E+05	2010,34	301,92	28,98	6,15	2491,55	318,36	289,60	286,80	286,80
40	160	10	329,62	95,67	11,67	1,66	178,00	67,94	73,20	73,00	73,00
40	160	1000	557,30	65,41	13,67	1,79	425,46	96,73	121,00	119,20	119,20
40	160	1E+05	554,16	60,68	13,15	2,06	465,29	119,64	118,20	118,20	118,20
40	280	10	755,75	208,32	22,02	3,72	771,66	291,27	142,20	142,00	142,00
40	280	1000	1300,39	111,47	24,98	1,89	1598,53	164,74	234,60	234,40	234,40

40	280	1E+05	962,25	556,32	24,33	2,71	1557,07	274,48	172,80	172,60	172,60
40	400	10	1411,90	375,00	32,98	4,05	1845,91	438,38	217,80	212,20	212,20
40	400	1000	2586,30	425,63	39,60	4,08	4027,01	652,09	367,60	366,00	366,00
40	400	1E+05	2654,44	295,26	39,50	2,80	4580,71	493,75	370,20	370,00	370,00

Tabla 4.4. Resultados computacionales de EEO1, EEO2 y L&P.

En la Figura 4.8 se muestra el tiempo de CPU de estos dos algoritmos con respecto al número de puntos extremos eficientes. Se observa que EEO2 es más rápido que L&P cuando el número de puntos solución no es muy grande. En este sentido, L&P es más robusto que EEO2 con respecto al numero de puntos extremos eficientes en la solución.

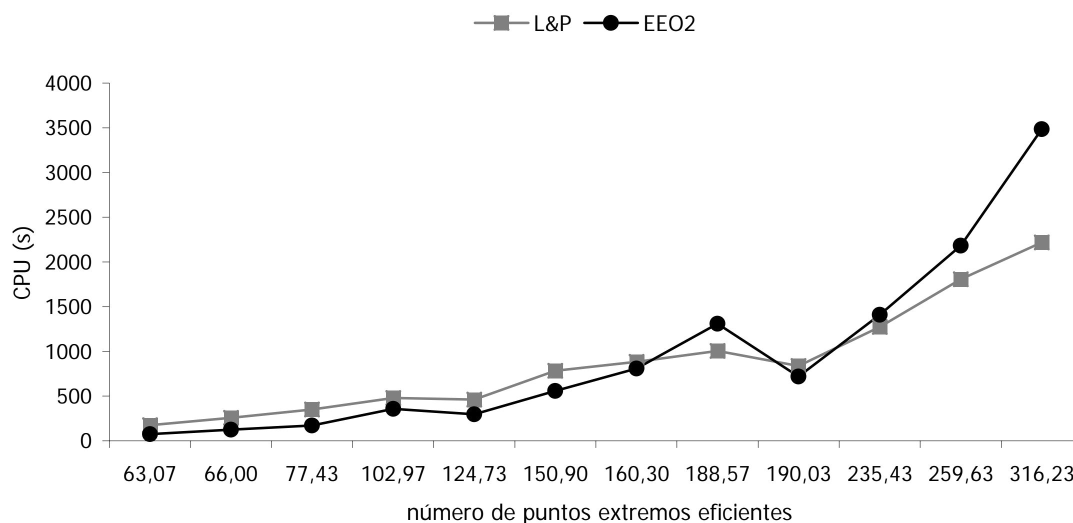


Figura 4.8. Tiempo de CPU frente a número de puntos extremos eficientes.

En la Figura 4.9, se muestra el crecimiento de EEO1 frente a la media de números de puntos extremos eficientes en el espacio objetivo. El tiempo de CPU aumenta con el número de estos. Evidentemente, no comparamos EEO1 con los otros algoritmos, pues este es significativamente más rápido.

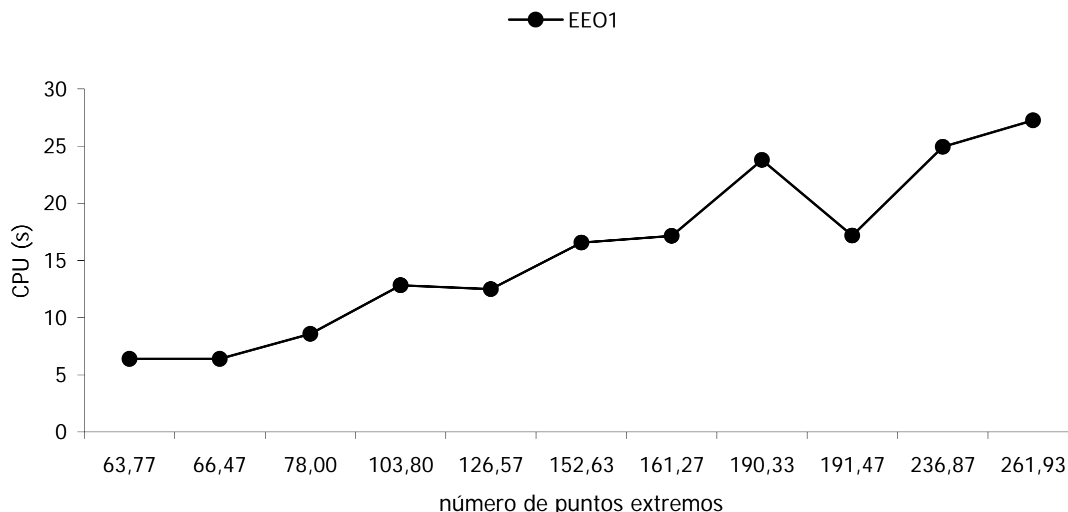


Figura 4.9.Tiempo de CPU frente a número de puntos extremos eficientes.

6. Caso entero

En esta sección daremos el algoritmo para resolver el problema FERB. Recordemos que habíamos denotado por $E_{ex}[f(X)]$ y por $E_{nex}[f(X)]$ al conjunto de puntos eficientes enteros extremos y no extremos, respectivamente, que están sobre la frontera eficiente en el espacio objetivo. Además, en el problema FERB decíamos que pueden existir puntos eficientes que no están sobre la frontera eficiente de $f(X)$. Estos puntos habrán de ser generados de manera separada. Nuestra intención es calcular todos los puntos eficientes del problema FERB, es decir, $E[f(X^I)]$.

El método que proponemos primero determina $E_{ex}[f(X)] \cup E_{nex}[f(X)]$, mediante la resolución del problema FRB, para, a continuación, determinar $E[f(X^I)] - (E_{ex}[f(X)] \cup E_{nex}[f(X)])$.

6.1 Obtención de todas las soluciones enteras que están sobre la frontera eficiente

El algoritmo para obtener $E_{ex}[f(X)] \cup E_{nex}[f(X)]$ se basa en el método de Sedeño-Noda y González-Martín [71] dado en la sección 3.1, que genera únicamente $E_{ex}[f(X)]$ para el caso continuo. Dicho

método comenzaba obteniendo el punto extremo eficiente $f(x^0)$ en el espacio objetivo. En el caso que nos ocupa, el siguiente paso consiste en encontrar el punto extremo eficiente en el espacio objetivo adyacente a $f(x^0)$ y, también, computar todos los puntos enteros eficientes que están entre ambos.

Recordamos que dado un punto extremo eficiente $f(x^i)$ en el espacio objetivo, se calcula el punto extremo eficiente adyacente $f(x^j)$ introduciendo en la base asociada del primero un conjunto de variables no básicas (sección 3.1).

Si $\theta = \min\left\{\frac{\overline{c}_{ij}^2}{\overline{c}_{ij}^1} : \overline{c}_{ij}^2 < 0 \ \forall(i,j) \in L, \frac{\overline{c}_{ij}^2}{\overline{c}_{ij}^1} : \overline{c}_{ij}^2 > 0 \ \forall(i,j) \in U\right\}$, entonces S es el conjunto que contiene todas las variable no básicas que alcanzan el valor de θ, es decir, las variables no básicas asociadas con una arista eficiente en el espacio objetivo.

Estamos interesados en generar todos los puntos enteros eficientes entre dos puntos extremos adyacentes $f(x^i)$ y $f(x^j)$. Para ello, sea $x_{uv} \in S$ una variable no básica del conjunto S y δ_{uv} las unidades de flujo en que cambia x_{uv} cuando entra en la base. Antes de añadir esta variable a la base, obtenemos $\delta_{uv} - 1$ puntos enteros eficientes obtenidos al incrementar (o decrementar) iterativamente en una unidad de flujo el valor de x_{uv}. Cada uno de estos puntos enteros eficientes es almacenado. Además, cuando x_{uv} cambia en δ_{uv} unidades de flujo, obtenemos una nueva base que es almacenada también. Si $\delta_{uv} = 0$ entonces actualizamos la base, los potenciales de los nodos y el correspondiente punto es almacenado para su examen futuro (base degenerada). A continuación el proceso se repite, con el siguiente arco no básico de S, hasta que este conjunto sea el vacío. En este momento, hemos obtenido un nuevo punto extremo eficiente y continuamos hasta alcanzar el punto extremo eficiente que optimiza la segunda función objetivo (el conjunto S coincide con el vacío para este punto).

Las anteriores ideas permiten desarrollar el siguiente algoritmo en el que se utilizan los índices de árbol *Pred*, *Depth* y *Thread* descritos en el capítulo 1, para mejorar las operaciones en

el proceso de pivoteo (ver Ahuja et al. [4], Cunningham [20]). Los procedimientos y el algoritmo tienen el siguiente esquema:

Algoritmo PEFE; {puntos enteros de la frontera eficiente}
  begin
  Obtener x^0, π^1, π^2 y (B,L,U) optimizando el primer objetivo mediante el método Simplex para redes;
  Sea Pred, Depth y Thread los índices del árbol;
  Almacenar (B,L,U), π^1, π^2 y $x = x^0$ como un punto extremo eficiente;
  Añade $\{x,(B,L,U),\pi^1,\pi^2\}$ a R; {R es el conjunto de puntos a examinar}
  while R <> $\emptyset$ **do**
    begin
    Sea $\{x,(B,L,U),\pi^1,\pi^2\}$ el primer entero eficiente en R;
    $R = R - \{x,(B,L,U),\pi^1,\pi^2\}$;
    Computar_Arcos_Entrantes(L, U, c, π^1, π^2, S);
    Computar_Nuevos_Puntos(x,B,L,U,π^1,π^2,Pred,Depth,Thread, S,R)
    end
  end.

procedure Computar_Arcos_Entrantes(L, U, c, π^1, π^2, S);
  begin
  $\overline{c}_{ij}^k = c_{ij}^k - \pi_i^k + \pi_j^k, \forall (i,j) \in A, k = 1,2;$
  S := $\emptyset$;
  Sea $\theta := \min\left\{ \frac{\overline{c}_{ij}^2}{\overline{c}_{ij}^1} : \overline{c}_{ij}^2 < 0 \quad \forall (i,j) \in L, \frac{\overline{c}_{ij}^2}{\overline{c}_{ij}^1} : \overline{c}_{ij}^2 > 0 \quad \forall (i,j) \in U \right\}$;
  Sea S el conjunto de arcos donde se alcanza el anterior mínimo
  end;

procedure Computar_Nuevos_Puntos(x, B, L, U, π^1, π^2,Pred, Depth, Thread, S, R);
  begin
  while S <> $\emptyset$ **do**
    begin
    Sea (i,j) el primer arco de S; S := S – (i,j);
    Nx := x; NB := B; NL := L; $N\pi^1 := \pi^1$; $N\pi^2 := \pi^2$;
    Npred := pred; Ndepth := depth; Nthread := Thread;
    Sea δ_{ij} las unidades de flujo en que (i,j) cambia con respecto al flujo Nx y la base NB y sea (p,q) el arco saliente;
    if $\delta_{ij} > 0$ **then**
      for t := 1 to δ_{ij} **do**
        begin
        Enviar una unidad de flujo a lo largo del ciclo de pivoteo definido por (i,j) y NB;
        Sea Nx el nuevo flujo;
        if t = δ_{ij} **then**

```
        begin
        Actualizar NB, NL, NU mediante (i,j) y (p,q);
        Actualizar Nπ1, Nπ2, NPred, NDepth, Nthread;
        Añadir {Nx,(NB,NL,NU),Nπ1,Nπ2} a R
        end;
      Almacenar NB, NL, NU, Nπ1, Nπ2 y Nx como entero
      eficiente
      end
  else
    begin
    Actualizar NB, NL, NU mediante (i,j) y (p,q);
    Actualizar Nπ1, Nπ2, NPred, NDepth, Nthread;
    Añadir {Nx,(NB,NL,NU),Nπ1,Nπ2} a R; {pivoteo degenerado}
    Almacenar NB, NL, NU, Nπ1, Nπ2 y Nx como entero eficiente
    end
  end
end;
```

Cada punto que almacena el algoritmo está caracterizado por el flujo *Nx*, la estructura fuertemente factible (*NB,NL,NU*) y los potenciales de los nodos $N\pi^1$, $N\pi^2$. Los índices del árbol *NPred*, *NDepth* y *NThread* pueden ser almacenados o calculados una vez que se conoce la base *NB*. El método mantiene una lista *R* que almacena los puntos extremos que faltan por examinar.

El procedimiento *Computar_Arcos_Entrantes* calcula el conjunto de arcos *S*. Estos arcos conforman la secuencia de pivoteos para alcanzar el punto extremo eficiente adyacente en el espacio objetivo. El punto extremo eficiente que es examinado en una iteración, está caracterizado por x, (B,L,U) y π^1, π^2. El procedimiento *Computar_Nuevos_Puntos* identifica todos los puntos eficientes enteros que son alcanzados a partir del punto extremo eficiente que actualmente se examina. Para realizar esto, en cada iteración el procedimiento selecciona un arco candidato (i,j) de *S*. Este procedimiento comienza con el flujo $Nx=x$ y, de manera iterativa, envía una unidad de flujo alrededor del ciclo definido por (i,j) y *NB* hasta que son enviadas $\delta_{ij}-1$ unidades de flujo. La orientación del ciclo de pivoteo depende de si el arco (i,j) pertenece al conjunto L o al U (ver Ahuja et al. [4], Cunningham [20]). Cada punto calculado por este procedimiento es almacenado como un punto eficiente entero. Todos estos puntos están caracterizados por la misma estructura (*NB,NL,NU*) y los mismos potenciales $N\pi^1$, $N\pi^2$. Cuando se incrementa x en δ_{ij} unidades de

flujo, se realiza la actualización de la base con el arco entrante (i,j) y el arco saliente (p,q) y, por lo tanto, se modifican (NB, NL, NU) y $N\pi^1$, $N\pi^2$. En este último caso, este punto se añade a R. Si δ_{ij} es igual a cero, entonces se realiza un pivoteo degenerado y se almacena la correspondiente estructura en R y en el conjunto de puntos enteros eficientes. A continuación se considera el siguiente arco candidato, empezando de nuevo a partir de x, (B,L,U), π^1, π^2 y el proceso se repite. Este procedimiento termina cuando el conjunto S está vacío y la estructura (NB, NL, NU) identifica un punto extremo eficiente en el espacio objetivo. Entonces, el algoritmo PEFE repite el proceso con el siguiente punto en R. Al finalizar el algoritmo se obtiene el conjunto de todos los puntos enteros sobre la frontera eficiente. Estos puntos se almacenan en una lista en orden no decreciente con respecto al valor alcanzado en el primer objetivo.

6.2 Obtención de los puntos enteros eficientes que no pertenecen a la frontera eficiente

Una vez generado el conjunto de puntos enteros $E_{ex}[f(X)] \cup E_{nex}[f(X)]$ sobre la frontera eficiente de $f(X)$, el siguiente paso consiste en determinar el conjunto $E[f(X^I)] - (E_{ex}[f(X)] \cup E_{nex}[f(X)])$; es decir, el conjunto de puntos eficientes enteros que no están sobre la frontera eficiente. Los puntos pertenecientes a este conjunto están en ln interior de los triángulos mostrados en la Figura 4.10. En esta figura, f^1, f^2 y f^3 corresponden a puntos extremos eficientes, y f_1^1, f_1^2, f_2^2 y f_3^2 pertenecen al conjunto $E_{nex}[f(X)]$.

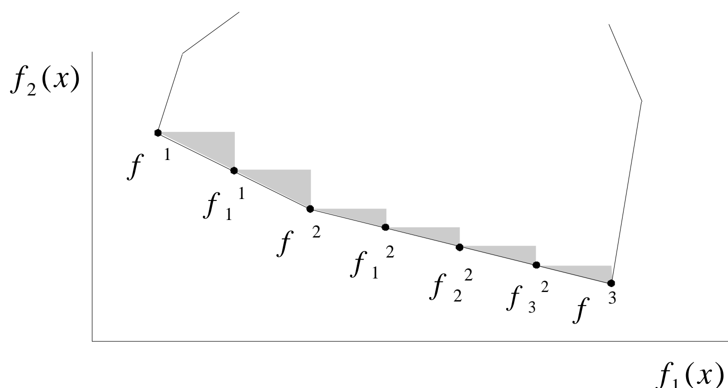


Figura 4.10. Región eficiente en el espacio objetivo del problema FERB

Dos puntos extremos sobre la frontera eficiente, $f(x^1)$ y $f(x^2)$ definen un triángulo rectángulo donde pueden ser localizados alguno puntos enteros eficientes del problema FERB. En la Figura 4.11 se puede observar esta situación. Sea x_{uv} la variable no básica que entra en la base de x^1 para obtener x^2. La base de x^1 junto con x_{uv} define un ciclo de pivoteo. Sea δ_{uv} igual a las unidades de flujo que pueden ser enviadas a través de este ciclo de pivoteo. De esta manera podemos obtener $\delta_{uv}-1$ puntos entre x^1 y x^2, enviando iterativamente una unidad de flujo a través de este ciclo. Llamaremos x_j^1 al punto obtenido a partir de x^1 después de enviar *j* unidades de flujo a lo largo del ciclo mencionado. En particular, $x_0^1 = x^1$ y $x_{\delta_{uv}}^1 = x^2$.

Dado $\bar{c}_{uv} = (\bar{c}_{uv}^1, \bar{c}_{uv}^2)$, definimos $\hat{c}_{uv} = \begin{cases} \bar{c}_{uv} & \text{si } x_{uv} \in L \\ -\bar{c}_{uv} & \text{si } x_{uv} \in U \end{cases}$, y, por lo tanto, $f(x^2) = f(x^1) + \delta_{uv}\hat{c}_{uv}$. Definimos $s_{ij} = \dfrac{\hat{c}_{ij}^2}{\hat{c}_{ij}^1}$ para todos los arcos no básicos (i,j). En particular s_{uv} corresponde a la pendiente de la recta que une los puntos $f(x^1)$ y $f(x^2)$.

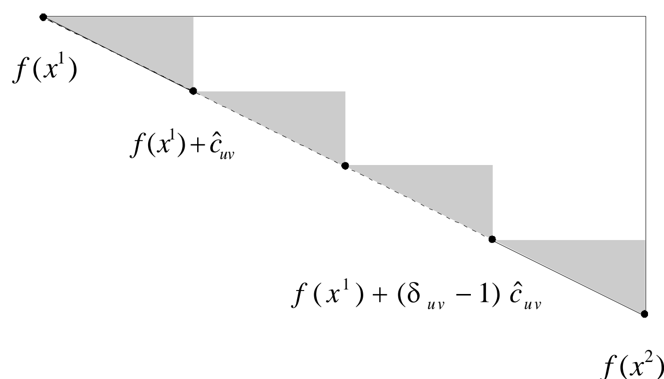


Figura 4.11. Triángulo definido por dos puntos extremos adyacentes en el espacio objetivo

Los puntos enteros eficientes del problema FERB se buscan en el interior de los triángulos cuyas hipotenusas son los segmentos $[f(x^1), f(x^1)+\hat{c}_{uv}], \ldots, [f(x^1)+(\delta_{uv}-1)\hat{c}_{uv}, f(x^2)]$. Sea (B^1, L^1, U^1) la estructura de árbol generador asociado con x^1. Por construcción, la misma estructura está asociada con los puntos x_j^1, con $j = 1, \ldots, \delta_{uv}-1$. A partir de cada uno de esos puntos, obtendremos los puntos enteros eficientes que pertenecen a estos triángulos.

Lema 4.1. *Sea* x^1 *un punto extremo eficiente del problema FRB. Sea* $R = \{(i,j) \in L^1 \text{ y } (i,j) \in U^1 : \hat{c}_{ij}^1 > 0 \text{ y } \hat{c}_{ij}^2 < 0 \text{ y } s_{ij} > s_{uv}\}$, *es decir, el conjunto de arcos no básicos con pendiente negativa y mayores que* s_{uv}. *Si para algún arco* (i,j) *de R es posible enviar una unidad de flujo a través del ciclo de pivoteo definido por este arco y la base de* x^1, *incrementando (decrementando)* x_{ij} *en una unidad, entonces la solución obtenida corresponde con una solución eficiente del problema FRB con* $l'_{ij} = l_{ij} + 1$ ($u'_{ij} = u_{ij} - 1$).

Demostración. Sea B^1 la base que está asociada con x^1 y sea $f(x^1)$ el correspondiente valor en el espacio objetivo. Consideraremos que $(i,j) \in L^1$ y supongamos que la cota inferior de este arco cambia por $l'_{ij} = l_{ij} + 1$. Entonces, x^1 no es una solución factible para el nuevo problema y, por lo tanto, es necesario enviar una unidad de flujo adicional a través de (i,j). Como (i,j) es un arco no básico en x^1, un incremento en la cota inferior l_{ij} no afecta a la configuración básica de este punto. Por lo tanto, para obtener factibilidad enviamos una unidad de flujo a través del ciclo de pivoteo definido por (i,j) y la base B^1, obteniendo el valor $f(x^1) + \hat{c}_{ij}$ en el espacio objetivo. Debido al hecho de que $f(x^1)$ era eficiente para el problema FRB original y a que el nuevo problema requiere una unidad de flujo adicional en el arco (i,j), el nuevo punto $f(x^1) + \hat{c}_{ij}$ es eficiente para el problema FRB con $l'_{ij} = l_{ij} + 1$. □

Claramente, el Lema 4.1 es aún cierto cuando se consideran simultáneamente cambios en las cotas de varios arcos de *R* en el problema FRB. Los puntos que se calculan de esta manera pueden ser eficientes para el problema FERB. En otras palabras, los puntos eficientes del problema FERB están incluidos en el conjunto de puntos enteros eficientes que resultan cuando se consideran diferentes problemas paramétricos del correspondiente problema FRB.

En el Lema 4.1 nos referimos a un conjunto especial R. La razón de la consideración de este conjunto R es dada en el siguiente lema.

Lema 4.2. *Cualquier combinación de arcos no básicos que contiene un arco no básico con pendiente positiva, está dominada por la misma combinación de arcos que no contiene el arco cuya pendiente es positiva.*

Demostración. Para ver esto, sea $\hat{c}_{pq}$ el coste reducido, previamente definido, de algún arco con pendiente positiva asociado con el punto eficiente x^1. Claramente, las dos componentes de $\hat{c}_{pq}$ son mayores que cero, y, por lo tanto, $f(x^1) < f(x^1) + \hat{c}_{pq}$. Por ello, el nuevo punto que se obtiene cuando una unidad (o más) de flujo es enviada a través del ciclo de pivoteo definido por (p,q), está dominado por x^1. La misma situación tiene lugar cuando se consideran varios arcos y uno o más de ellos tienen pendiente positiva. □

En R los arcos no básicos (i,j) con pendiente positiva (lema 4.2) y los arcos tales que $\hat{c}_{ij}^1 < 0$ y $\hat{c}_{ij}^2 > 0$ (con pendiente negativa) no se tienen en cuenta, ya que el método que proponemos genera las soluciones eficientes en orden no decreciente de $f_1(x)$. Esta construcción del conjunto R, junto con una regla para eliminar los arcos de R, garantizará que en nuestro método cada solución eficiente del problema FERB sea generada una sola vez.

Teorema 4.8. *Sea (i,j) un arco de R, tal que $(i,j) \in L^1$ ($(i,j) \in U^1$). Entonces un incremento (decremento) en una unidad de la variable x_{ij}, enviando una unidad de flujo a lo largo del ciclo de pivoteo definido por el arco (i,j), implica uno de los dos siguientes casos:*

(i) El nuevo punto está en el interior de algún triángulo cuya hipotenusa esta definida por los puntos $f(x_j^i)$ y $f(x_{j+1}^i)$ para

algún j y algún x^i *con* $i \geq 1$. *Este nuevo punto puede ser eficiente para el problema FERB.*

(ii) *El nuevo punto está en el interior del triángulo definido por dos puntos extremos eficientes adyacentes, pero fuera de los triángulos definidos por los puntos enteros eficientes. Este nuevo punto no es eficiente.*

Las definiciones, lemas y teoremas anteriores nos dan las herramientas para idear un algoritmo que obtenga todas las soluciones enteras eficientes del problema FERB. Una cuestión importante que se ha de tener en cuenta en el esquema del algoritmo es que este no debe generar soluciones dominadas. Esto impedirá incorporar herramientas en el método para eliminar con posterioridad soluciones dominadas. Para esto, cada punto entero eficiente que es calculado por el algoritmo tiene asociado un conjunto de arcos candidatos R. Los arcos $(i,j) \in R$ se almacenan en una lista en orden no decreciente de $\hat{c}^1_{ij}$. Por lo tanto, el punto entero eficiente y, que es calculado en cada iteración del método, se corresponde con el punto entero eficiente que tiene el valor más pequeño del primer objetivo. En otras palabras, $f_1(y) = \min_{x^i} \left(f_1(x^i) + \hat{c}^1_{ij} \right)$, donde x^i recorre todos los puntos enteros eficientes que han sido calculados por el algoritmo cuyo conjunto asociado es $R \neq \phi$. Si existe más de un punto y, se elige el punto con el menor valor de $f_2(y)$ (todos los otros están dominados por este punto). De esta manera, cuando un punto candidato se examina, podemos fácilmente detectar si este punto es no eficiente, debido a que está dominado por algún punto entero eficiente calculado con anterioridad. La Figura 4.12 muestra esta situación. Sean x^1, x^2, x^3 y x^4 los puntos enteros eficientes que se conocen actualmente. Cada uno de ellos tiene asociado su correspondiente conjunto R. El punto que es elegido entre los puntos p^1, p^2, p^3, p^4, p^5, p^6 y p^7 es el punto p^1. Se puede observar que p^2, p^3 y p^6 son puntos dominados. Estos puntos no serán generados por el algoritmo, ya que p^2, p^3 están dominados por $f(x^2)$ y en el proceso son calculados p^4, p^5 antes de que el punto p^6 sea examinado.

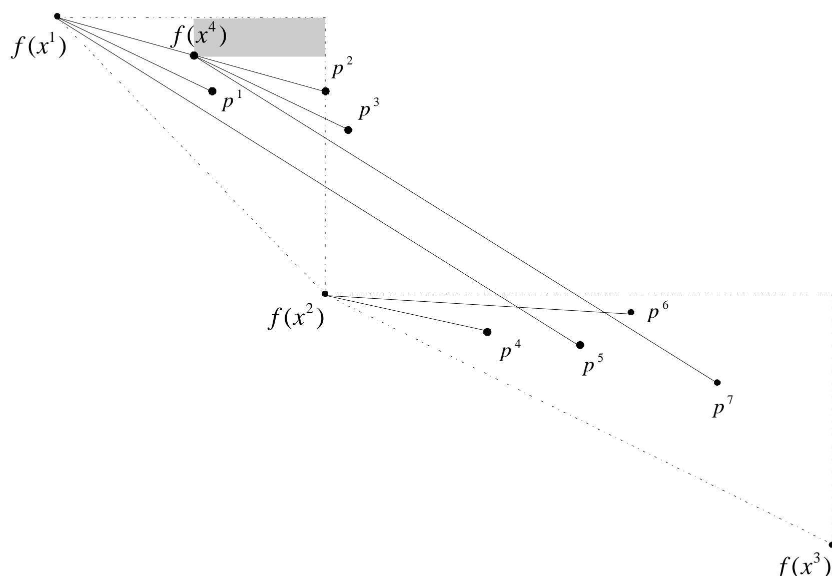


Figura 4.12. Punto entero eficiente calculado en una iteración del método

Otra cuestión importante a considerar es que el método genere cada solución eficiente sólo una vez. Ya hemos explicado algunos detalles para resolver este problema. Por ejemplo, la consideración del conjunto *R* y que las soluciones enteras eficientes sean calculadas en orden no decreciente según el valor del primer objetivo, con la excepción de las soluciones sobre la frontera eficiente. Además, es necesario asignar de manera adecuada el conjunto *R* para aquellos puntos eficientes que el algoritmo obtiene. Para realizar esto, cuando se calcula una solución eficiente x^2 a partir de x^1, después de enviar una unidad de flujo a través del ciclo pivote definido por algún arco $(i,j) \in R^1$, el conjunto R^2 de x^2 coincide con el conjunto R^1 y hacemos $R^1 = R^1 - (i,j)$. La Figura 4.13 muestra esta situación. Los puntos son generados en el orden p^1, p^2, p^3, p^4 a partir del conjunto de arcos $R = \{a,b\}$ del punto $f(x^1)$. El punto p^1 se genera a partir de $f(x^1)$ y hacemos $R^1 = \{a,b\}$ y $R = \{b\}$. El punto p^2 se obtiene de $f(x^1)$, y en este caso $R^2 = \{b\}$ y $R = \varnothing$. Continuando el esquema, los puntos p^3 y p^4 se obtienen a partir de p^1, haciendo $R^3 = R^1 = \{a,b\}$ y $R^4 = R^1 = \{b\}$. El punto p^4 podría ser generado a partir de p^2 si el arco *a* (línea de puntos) perteneciese a R^2, pero la construcción propuesta garantiza que esto no ocurra.

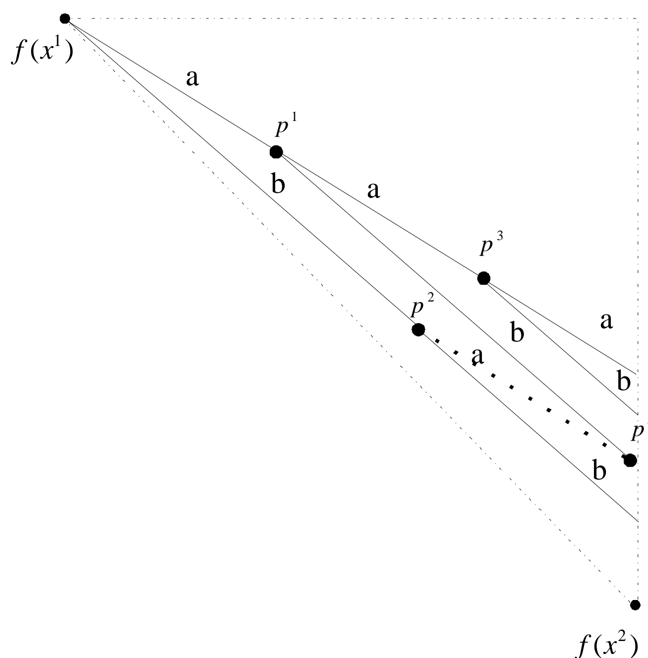


Figura 4.13. Construcción del conjunto R para cada punto entero eficiente

Teorema 4.9. *Sea x una solución entera eficiente del problema FERB y R su correspondiente conjunto de arcos. Sea* x^q *el punto entero eficiente que se obtiene a partir de x mediante el subconjunto de arcos* $Q \subseteq R$. *Si se elige en cada iteración el arco* $(i,j) \in R$ *con el valor más pequeño de* $\hat{c}^1_{ij}$ *y, a continuación, R es actualizado por* $R = R - (i,j)$, *entonces se obtiene* x^q *sólo una vez.*

Demostración. Sea $Q = \{a^1, a^2, ..., a^p\}$ el subconjunto de arcos. Sin pérdida de generalidad, podemos suponer que los arcos en *R* y en *Q* están enumerados en orden no decreciente de $\hat{c}^1$. Después de que se elige el primer arco $b^1 \in R$, puede ocurrir alguno de los dos siguientes casos:

(i) $b^1 \neq a^1$. En este caso se obtiene un punto x^{b^1} con $R^1 = R$ y, a continuación, $R = R - b^1$. Es evidente que x^q nunca se obtendrá a partir de x^{b^1}.

(ii) $b^1 = a^1$. En este caso se obtiene un punto x^{a^1} con $R^1 = R$ y $R = R - a^1$. Por lo tanto, x^q puede ser obtenido a partir de x^{a^1}, pero no a partir de cualquier otro punto *y* obtenido a partir de *x*, debido a que el conjunto de arcos asociado con *y* no contiene el conjunto *Q*, es decir, $R^y : Q \not\subset R^y = R$.

Ahora bien, si consideramos que $x = x^{a^1}$ y $Q = Q - a^1$ y repetimos el anterior esquema, se prueba que x^q se obtiene a partir de x después de elegir la secuencia de arcos $a^1, a^2, \ldots, a^p$. Además, esta secuencia se construye una sola vez. □

A continuación discutiremos las diferencias entre nuestro método y el método de Lee y Pulat [54]. Lee y Pulat introducen un método para generar todos los puntos enteros eficientes del problema FERB mediante la realización de análisis paramétrico sobre el correspondiente problema FRB. En su método, la cota inferior (o superior) de un arco no básico es perturbada para cada punto de la frontera eficiente. En este caso se consideran todos los arcos no básicos que pertenecen a cada solución de la frontera eficiente. En este sentido, nuestro método coincide con el anterior. Sin embargo, su método no calcula todas las soluciones del problema FERB, debido a que, para una solución fijada de la frontera eficiente, no considera perturbaciones sobre varios arcos no básicos de manera simultánea. En nuestro método se realizan perturbaciones sobre varios arcos no básicos a la vez. Por ejemplo, para el ejemplo hipotético de la Figura 4.13, el método de Lee y Pulat nunca calcula el punto p^4, debido a que este punto se obtiene a partir de x^1 cuando las cotas inferiores (o superiores) de los arcos $\{a,b\}$ son perturbadas simultáneamente en una unidad. Además, en el método de Lee y Pulat las soluciones no son calculadas en orden no decreciente de $f_1(x)$. Por lo tanto, pueden ser calculadas soluciones dominadas que posteriormente han de ser eliminadas. Nuestro método no calcula soluciones dominadas. A continuación, comentaremos en detalle la implementación del algoritmo.

6.3 Algoritmo

Las ideas previas permiten desarrollar un algoritmo para obtener todos los puntos enteros eficientes del problema FERB de tal manera que cada punto eficiente se genera una sola vez, impidiendo la generación de puntos dominados. Cada punto entero eficiente es caracterizado por un flujo *x,* el vector $f(x)$, los índices del árbol *Pred*, *Depth* que permiten realizar los envíos de flujo y el

conjunto de arcos R. Para cada $(i,j)\in R$ se almacena el vector $\bar{c}_{ij}$. Los índices del árbol *Pred*, *Depth* y los costes reducidos $\bar{c}_{ij}$ son heredados de los puntos enteros sobre la frontera eficiente del problema FERB, los cuales se obtienen mediante el algoritmo PEFE. El algoritmo mantiene una lista denominada *LIST* que almacena los puntos enteros eficientes calculados. En *LIST* se almacenan los puntos en orden no decreciente de valores del primer objetivo. El esquema del algoritmo es dado a continuación.

El algoritmo comienza almacenando en *LIST* todos los puntos enteros sobre la frontera eficiente del problema FERB, en orden no decreciente de valores del primer objetivo. Cada uno de esos puntos tiene asociada la estructura $\{x,\ f(x),\ Pred,\ Depth,\ R\}$. Los arcos en R se almacenan en orden no decreciente de $\left|\bar{c}^1_{ij}\right|$. El procedimiento *Arco_Candidato* devuelve el punto con el valor más pequeño $f_1(x)+\hat{c}^1_{ij}$ de entre los puntos enteros eficientes en *LIST*. Si $f(x)+\hat{c}_{ij}$ no está dominada, entonces pueden ocurrir dos casos:

```
Algoritmo PEE; {puntos enteros eficientes}
  begin
  Almacenar en LIST todos los puntos enteros de la frontera
  eficiente obtenidos por el algoritmo PEFE;
  repeat
    Arco_Candidato(x,R,Pred, Depth,f(x));
    if existe un arco candidato then
      begin
      Sea (i,j) el primer arco de R;
      if c̄¹_ij < 0 then ĉ_ij = -c̄_ij else ĉ_ij = c̄_ij;
      if f(x)+ ĉ_ij no está dominado then
        begin
        if c̄¹_ij > 0 then
          begin
          Usar Pred y Depth para identificar el camino P_ji de j
          a i;
          Sea δ_ij la mínima cantidad de flujo que puede ser
          enviada a través del ciclo {P_ji ,(i,j)} con respecto al
          flujo x;
          if δ_ij > 0 then
            begin
            Npred := Pred; Ndepth := Depth; Nx := x;
            Envía una unidad de flujo a lo largo del ciclo
            {P_ji ,(i,j)}; sea Nx el nuevo flujo;
            Almacena en LIST {f(x)+ĉ_ij , Nx, Npred, Ndepth,R}
            end
```

```
          end
        else
          begin
          Usar Pred y Depth para identificar el camino P_ij de i
          a j;
          Sea δ_ij la mínima cantidad de flujo que puede ser
          enviada a través del ciclo {P_ij,(j,i)} con respecto al
          flujo x;
          if δ_ij > 0 then
            begin
            Npred := Pred; Ndepth := Depth; Nx := x;
            Envía una unidad de flujo a lo largo del ciclo
            {P_ij,(j,i)}; Sea Nx el nuevo flujo;
            Almacena en LIST {f(x)+ĉ_ij, Nx, Npred, Ndepth,R}
            end
          end
      R := R - (i,j)
      end;
  until (no exista un arco candidato)
  end.
```

Caso a): Si $\overline{c}_{ij}^{1} > 0$ (el arco proviene de estar en su cota inferior) entonces, si es posible, se envía una unidad de flujo a lo largo del ciclo definido por el arco (i, j) y P_{ji}.

Caso b): Si $\overline{c}_{ij}^{1} < 0$ (el arco proviene de estar en su cota superior) entonces, si es posible, se envía una unidad de flujo a lo largo del ciclo definido por (j,i) y P_{ij}.

A continuación se almacena el nuevo punto calculado en *LIST*. El algoritmo finaliza cuando no se determina un arco candidato, es decir, cuando todos los conjuntos R están vacíos.

Usamos un puntero denominado *Ultimo_Punto* para mejorar el tiempo de ejecución del procedimiento *Arco_Candidato*. *Ultimo_Punto* apunta al primer punto eficiente en *LIST* cuyo conjunto R es no vacío. Es evidente que un punto entero eficiente, cuya posición en *LIST* es anterior a *Ultimo_Punto*, nunca se alcanza. Además, para un punto entero eficiente de *LIST* únicamente el primer arco en R es examinado, con la excepción de que en el proceso de búsqueda este arco implica una solución dominada. En este último caso, este arco se borra y se examina el siguiente arco en R.

Cada dos puntos consecutivos en `LIST` definen un triángulo en el espacio objetivo. Sea T el triángulo que incluye $f(x)+\hat{c}_{ij}$. Cuando el algoritmo examina si $f(x)+\hat{c}_{ij}$ es eficiente, recorre `LIST` hasta que se identifica el triángulo T. El método recorre `LIST` comenzando en la posición que ocupa el punto x en `LIST`, es decir, la dada por el procedimiento `Arco_Candidato` para mejorar el tiempo de ejecución de esta operación. Además, después que el algoritmo realiza esta operación, conocemos la posición de almacenamiento en `LIST` del nuevo punto calculado.

6.4 Complejidad teórica del algoritmo

La complejidad del algoritmo depende del número de soluciones enteras eficientes del problema FERB. El algoritmo examina cada uno de los arcos en R de todas las soluciones enteras eficientes. En el caso peor, el cardinal del conjunto R es O(m-n). Sea $C=\max\{\left|c_{ij}^1\right| / \forall(i,j)\in A\}$. El procedimiento `Arco_Candidato` recorre `LIST` comenzando en `Ultimo_Punto`. Supongamos que `Ultimo_Punto` apunta a la solución entera eficiente y. Sea z el primer punto eficiente en `LIST` tal que $f_1(z)>f_1(y)+\hat{c}_y^1$. Es evidente que entre los puntos y y z hay, a lo sumo, $\hat{c}_y^1$ (coste reducido asociado a y) puntos. Por lo tanto, como $\hat{c}_y^1\leq(2n+1)C$ (ver Ahuja et al. [4]), el tiempo de ejecución del procedimiento `Arco_Candidato` es O(nC).

El test de eficiencia obliga al método a recorrer `LIST` desde el punto x que devuelve el procedimiento `Arco_Candidato`. Esta operación tiene la misma complejidad que el procedimiento `Arco_Candidato`, es decir, el test está acotado por O(nC).

La operación de actualización del flujo mediante los índices del árbol *Pred* y *Depth* está acotada por O(n). La complejidad de la operación que realiza la copia de la estructura de un punto es O(m). Finalmente, la operación de almacenamiento (inserción en orden) de la estructura del nuevo punto entero eficiente en `LIST` es O(1), debido a que su posición en `LIST` se conoce después de la realización del test de eficiencia. Por todo esto, la complejidad del

algoritmo PEE es O($\left|E\left[f(X^I)\right]\right|(m-n)(nC+nC+n+m)$) y, considerando únicamente los términos dominantes, es O($m(nC+m)\left|E\left[f(X^I)\right]\right|$).

6.5 Un ejemplo

Consideraremos el ejemplo introducido por Lee y Pulat [54] dado en la Figura 4.14.

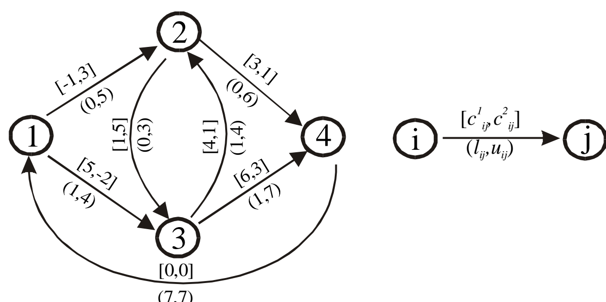


Figura 4.14. Red del ejemplo

Primero el algoritmo genera el conjunto de soluciones enteras sobre la frontera eficiente del problema FRB. En la Tabla 4.5 se muestran estos puntos. El algoritmo PEFE calcula x^1, x^2 y x^3, donde x^1 y x^3 son puntos extremos y x^2 no lo es. Los puntos x^2 y x^3 se obtienen a partir del punto x^1 cuando el arco (3,2) entra en su base. El mayor incremento en el ciclo definido por (3,2) es 2 unidades de flujo. Un envío de una unidad alrededor de este ciclo genera el punto x^2 y un envío de dos unidades genera el punto x^3. La Tabla 4.5 muestra el conjunto de arcos candidatos R para cada punto usado por el algoritmo PEE. También se muestran los flujos de los arcos y los costes reducidos. El siguiente paso en el algoritmo consiste en determinar los puntos enteros eficientes que no están sobre la frontera eficiente.

	(1,2)	*(1,3)*	*(2,3)*	*(2,4)*	*(3,2)*	*(3,4)*	*(4,1)*	f_1	f_2	*R*
$(x_{ij}, \hat{c}^1_{ij}, \hat{c}^2_{ij})$	(5,0,0)	(2,0,0)	(0,5,-10)	(6,0,0)	(1,10,-4)	(1,9,-3)	(7,2,4)	33	21	{(3,4)}
$(x_{ij}, \hat{c}^1_{ij}, \hat{c}^2_{ij})$	(4,0,0)	(3,0,0)	(0,5,-10)	(6,0,0)	(2,10,-4)	(1,9,-3)	(7,2,4)	43	17	{(3,4)}
$(x_{ij}, \hat{c}^1_{ij}, \hat{c}^2_{ij})$	(3,0,0)	(4,10,-4)	(0,5,6)	(6,0,0)	(3,0,0)	(1,1,-1)	(7,2,4)	53	13	{ }

Tabla 4.5. Soluciones enteras sobre la frontera eficiente

La Tabla 4.6 muestra la ejecución del algoritmo PEE para el ejemplo. En esta tabla, aparece *LIST* conteniendo todas las soluciones enteras eficientes calculadas. Estos puntos son

enumerados en orden no decreciente de valores de f_1. La Tabla 4.6 muestra el punto p obtenido por el procedimiento `Arco_Candidato`, y el conjunto de arcos R_P. Además, en esta tabla aparece el nuevo punto calculado NP por el algoritmo, su correspondiente imagen y el conjunto R_{NP} asociado. Los valores de los flujos de cada punto se muestran en la Tabla 4.7. Al principio del algoritmo, p es x^1 y el primer arco en R_P es (3,4), el cual está en su cota inferior. Se obtiene el punto x^4, enviando una unidad de flujo a lo largo del ciclo definido por este arco. Este punto se almacena en `LIST` y su conjunto de arcos R_{NP} coincide con el conjunto del punto x^1. A continuación, el arco (3,4) es eliminado del conjunto R_P asociado con x^1. El algoritmo continúa eligiendo $p=x^4$. A partir de este punto, se obtiene el punto x^5 mediante un envío adicional de una unidad de flujo a través del ciclo definido por el arco (3,4). Luego se elige el punto $p=x^2$, obteniendo el punto x^6. Más tarde, el algoritmo elige los puntos $p=x^5$ y $p=x^6$. A partir de estos dos puntos es imposible enviar una unidad de flujo adicional a través del ciclo definido por (3,4), debido a que el arco (1,3) está en su cota superior. Finalmente, como todos los conjuntos R están vacíos el método termina.

LIST	*p*	R_p	*NP*	F_1	f_2	R_{NP}
$\{x^1, x^2, x^3\}$	x^1	{(3,4)}	x^4	42	18	{(3,4)}
$\{x^1, x^4, x^2, x^3\}$	x^4	{(3,4)}	x^5	51	15	{(3,4)}
$\{x^1, x^4, x^2, x^5, x^3\}$	x^2	{(3,4)}	x^6	52	14	{(3,4)}
$\{x^1, x^4, x^2, x^5, x^6, x^3\}$	x^5	{(3,4)}				
$\{x^1, x^4, x^2, x^5, x^6, x^3\}$	x^6	{(3,4)}				

Tabla 4.6. Ilustración de la ejecución del algoritmo PEE para el ejemplo

	(1,2)	*(1,3)*	*(2,3)*	*(2,4)*	*(3,2)*	*(3,4)*	*(4,1)*
x^4	4	3	0	5	1	2	7
x^5	3	4	0	4	1	3	7
x^6	3	4	0	5	2	2	7

Tabla 4.7. Flujos de los puntos generados por el algoritmo PEE

6.6 Resultados computacionales

PEE es un código PASCAL que ha sido ejecutado en una estación de trabajo HP9000/712 a 60 MHZ. Los problema test fueron generados usando NETGEN [52]. Los valores de los costes de

la primera función objetivo son dados por NETGEN y los costes de la segunda función objetivo fueron generados uniformemente en el intervalo [-1000,1000] (todos estos valores son enteros). La Tabla 4.8 muestra los niveles del número de nodos (n), número de arcos (m), ratio m/n y la máxima capacidad de los arcos (U):

n	m	m/n	U
10	20, 30, 40	2, 3, 4	20, 40, 60
15	30, 45, 60	2, 3, 4	20, 40, 60
20	40, 60, 80	2, 3, 4	20, 40, 60
25	50, 75, 100	2, 3, 4	20, 40, 60

Tabla 4.8. Número de nodos, arcos, ratio m/n y U

Mediante las combinaciones de nodos (n), arcos (m) y capacidad máxima (U), obtenemos 36 parámetros de red. Para cada una de las combinaciones anteriores, hemos generado 5 replicas mediante NETGEN y las siguientes semillas: 12345678, 36581249, 23456183, 46545174, 35826749. Por lo tanto, el número de casos particulares de estudio fueron 180.

En la Tabla 4.9, se muestran los resultados experimentales del algoritmo PEE. En esta tabla aparece el número de nodos (n), de arcos (m), la capacidad máxima de los arcos (U), la media del tiempo de CPU en segundos ($mCPU$), la media de todas las soluciones enteras eficientes ($mPEE$) para el problema FERB, la media de las soluciones enteras de la frontera eficiente ($mPEFE$) y la media de las soluciones eficientes que no están sobre la frontera eficiente ($mPENFE$).

n	*m*	*U*	*mCPU(s)*	*mPEE*	*mPENFE*	*mPEFE*
10	20	20	0,36	66,4	43	23,4
10	20	40	0,826	161,6	113,4	48,2
10	20	60	1,48	268,2	161,4	106,8
10	30	20	0,274	45,4	22,6	22,8
10	30	40	0,484	95,4	40,2	55,2
10	30	60	0,598	132	61,2	70,8
10	40	20	0,35	60,8	38,2	22,6
10	40	40	1,058	189,2	130,4	58,8
10	40	60	0,966	177,2	102,8	74,4
15	30	20	2,406	337,4	251,4	86
15	30	40	18,174	938,6	784	154,6
15	30	60	38,62	1565	1295,2	269,8
15	45	20	3,254	350	282,6	67,4
15	45	40	46,12	1620	1452,2	167,8
15	45	60	130,924	2325	2029,8	295,2
15	60	20	5,368	469,4	385	84,4
15	60	40	20,298	1197,8	1036,8	161
15	60	60	98,754	2772,6	2436	336,6
20	40	20	3,342	319,6	255,4	64,2
20	40	40	6,448	706,2	563	143,2
20	40	60	6,272	699,2	535,8	163,4
20	60	20	14,432	806,2	698,6	107,6
20	60	40	60,918	1880	1673,6	206,4
20	60	60	107,088	2602,2	2276	326,2
20	80	20	38,754	1416	1298,6	117,4
20	80	40	175,348	2813,2	2617,6	195,6
20	80	60	194,054	3508,2	3215,2	293
25	50	20	8,696	634,2	547,4	86,8
25	50	40	173,392	2276,4	2050,4	226
25	50	60	95,52	2134,4	1873,8	260,6
25	75	20	56,414	1402,6	1283,6	119
25	75	40	80,456	2178	1936	242
25	75	60	404,646	4314,2	3910,2	404
25	100	20	190,636	2735,6	2558	177,6

25 100 40	370,734	4255,8	3937,4	318,4	25 100 60	662,786	6401,2	5916,2	485

Tabla 4.9. Media del tiempo de CPU y del número de soluciones enteras eficientes

De la Tabla 4.9, cuando se suman todas las soluciones, se extrae que el número de soluciones que no están en la frontera eficiente es aproximadamente el 88,78 por ciento del total. Además, el número de estas soluciones aumenta cuando incrementa el tamaño de las redes. La Figura 4.15 muestra el crecimiento de la media del tiempo de CPU en segundos (*mCPU*), de la media de soluciones en la frontera eficiente (*mPEFE*) y de la media de soluciones que no pertenecen a la frontera eficiente (*mPENFE*), para cada uno de los valores de máxima capacidad. Esta figura muestra que el número de soluciones del problema FERB incrementa cuando la máxima capacidad incrementa y que *mPENFE* incrementa más rápido que *mPEFE*. Esta última cuestión justifica los pequeños niveles de U. Si U es muy grande, entonces el número de soluciones es muy grande y, por tanto, se requiere más tiempo de CPU.

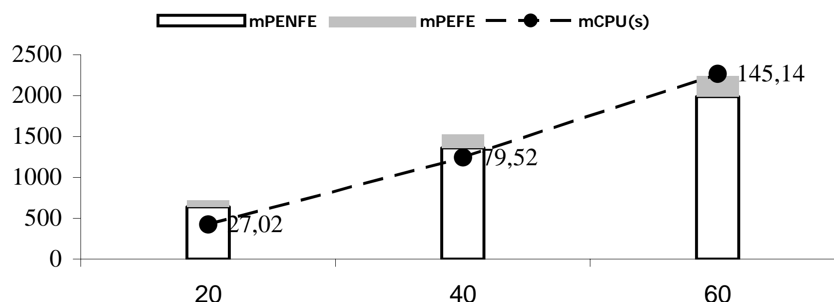


Figura 4.15. Crecimiento del tiempo CPU y de las soluciones enteras eficientes que están y no están sobre la frontera eficiente con respecto a U.

Bibliografía

[1] Ahuja, R, Orlin, J. B., Tarjan, R. E., "Improved Time Bounds for the Maximum Flow Problem", *SIAM Journal of Computing* 18 (1989) 939-954.

[2] Ahuja, R., Kodialam, M., Mishra, A. K., Orlin, J. B., "Computational investigations of maximum flow algorithms", *European Journal of Operational Research* 97 (1997) 509-542.

[3] Ahuja, R., Magnanti, T., Orlin, J. B., *Network Flows. In Optimization*. Handbooks in Operations Research and Management Science 1 (1989) 211-369, G.L.Nemhauser, A.H.G. Rinnooy Kan, M.J.Todd (eds.), North Holland.

[4] Ahuja, R., Magnanti, T., Orlin, J. B., *Network Flows*. Prentice-Hall, inc (1993).

[5] Ahuja, R., Orlin, J. B., "Distance-Directed Augmenting Path algorithms for Maximum Flow and Parametric Maximum Flow problems", *Naval Research Logistics Quarterly* 38 (1991) 413-430.

[6] Ahuja, R., Orlin, J. B.," A fast and simple algorithm for the Maximum Flow Problem", *Operations Research* 37 (1989) 748-759.

[7] Anderson, R. J., Setubal, J. C.,"Parallel and sequential implementations of maximum-flow algorithms", in: D.S. Johnson and C.C. McGeoch (eds), *Network Flows and Matching*: First DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 12 (1993) American Mathematical Society.

[8] Anderson, V. L., Mclean, R. A., *Design of Experiments. A realistic approach*. Marcel Dekker (1974).

[9] Aneja, Y. P. and Nair K. P., "Bicriteria Transportation Problem", *Magnagement Science*, Vol. 25, nº 1, January (1979).

[10] Bazaraa, M. S., Jarvis, J. J., Sherali, H. D., "Linear Programming and Network Flows", 2nd ed. Wiley, New York (1990).

[11] Bertsekas, D. P., Tseng, P., " Relaxation methods for minimun cost ordinary and generalized network flow problems", *Operations Research* 36 (1988) 93-114.

[12] Box, G. E. P., Cox, D. R., "An Analysis of Transformation", *Journal of the Royal Statistic Soc*. 26 (1964) 211-252.

[13] Bradley, G., Brown, G., Graves, G., "Design and implementation of large scale primal transshipment algorithms", *Managament science* 21 (1977) 1-38.

[14] Bryson, N., "Parametric programming and lagrangian relaxation: The case of the network problem with a single side-constraint", *Computers and*

Operations Research, 18 (2) (1991), 129-140.

[15] Busaker, R. G., Gowen, P. J., "A procedure for determining minimal-cost network flow patterns", *ORO Technical Report 15, Operation Research Office*, Johns Hopkins University, Baltimore, Md (1961).

[16] Calvete, H. I. and Mateo, P. M., "An approach for the network flow problem with multiple objectives", *Computers and Operations Research,* 22 (9) (1995), 971-983.

[17] Chang, M. D., Chen, C. J., "An improved primal simplex variant for pure processing networks", *ACM transactions on Mathematical Software* 15 (1989) 64-78.

[18] Cheriyan, J., Hagerup, T., Mehlhorn, K., "An O($n^3/\log n$)-Time Maximum-Flow Algorithm", *SIAM Journal of Computing* 25 (1996) 1144-1170.

[19] Cheriyan, J., Maheshwari, S.N., "Analysis of Preflow Push Algorithms for Maximum Network Flow", *SIAM Journal of Computing* 18 (1989) 1057-1086.

[20] Cunningham, W. H., "A Network Simplex Method", *Mathematical Programming* 11 (1976), 105-106.

[21] Cunningham, W. H., "Theoretical properties of the network simplex method", *Mathematics of Operations Research,* 4 (1979), 196-208.

[22] Current, J. and Min, H., "Multiobjective design of transportation networks: taxonomy and annotation", *European Journal of Operational Research* 26 (1986), 187-201.

[23] Dantzig, G. B., "Application of the simplex method to a transportation problem", In *activity Analysis and Production and Allocation*, edited by T. C. Koopmans. Wiley, New York (1951) 359-373.

[24] Derigs, U., Meier, W., "Implementing Goldberg's Max-Flow-algorithm a Computational Investigation", *Methods and Models of Operations Research* 33 (1989) 383-403.

[25] Dinic, E. A., "Algorithms for solution of a problem of Maximum Flow in Networks with power estimation", *Soviet Mathematical Doklady* 11 (1970) 1277-1280.

[26] Edmonds, J, Karp, R. M., "Theoretical Improvements in Algorithmic Efficiency of Network Flow problems", *Journal of ACM* 19 (1972) 248-264.

[27] Fernández-Baca, D., Martel, C. U., "On the Efficiency of Maximum_Flow Algorithms on Networks with Small Integer Capacities", *Algorithmica* 4 (1989) 173-189.

[28] Ford, L. R., Fulkerson, D. R., "Maximal Flow through a Network", *Canadian Journal of Mathematics* 8 (1956) 399-404.

[29] Ford, L.R, Fulkerson, D. R., "A primal-dual algorithm for the capacitated Hitchcock problem", *Naval Research Logistics Quarterly* 4 (1957) 47-54.

[30] Fulkerson, D. R., "An out-of-kilter method for minimal cost flow problems", *SIAM journal on Applied Mathematics* 9 (1961) 12-27.

[31] Gabow, H. N., "Scaling Algorithms for Network Flow Problems", *Journal of computer and System Sciences* 31 (1985) 148-168.

[32] Gal, T., *Postoptimal analyses, parametric programming and related topics*. McGraw Hill, inc (1979).

[33] Gass, S. and Saaty, T., "The computational algorithm for the parametric objective function", *Naval Research Logistics Quarterly* 2 (1955) 39-45.

[34] Geoffrion, A.,M., "Solving bicriterion mathematical programs", *Operations Research* 15 (1967), 39-54.

[35] Glover, F., Karney, D., Kligman, D., Napier, A., "A Computational study on star procedures, basis change criteria and solution algorithms for transportation problem", *Management Science* 20 (1974), 793-813.

[36] Glover, F., Karney, D., Klingman, D., "Implementation and computational comparisons of primal, dual and primal-dual computer codes for minimum cost network flow problem", *Networks* 4 (1974) 191-212.

[37] Goldberg, A. V, Tarjan, R. E., " A new approach to the Maximum Flow problem", Proc. 18th ACM Symp. on the Theory of Computation (1986) 136-146.

[38] Goldberg, A. V., " A New Max-Flow Algorithm", Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT, Cambridge, MA (1985).

[39] Goldfarb, D., Hao, J., "Anti-Stalling Pivot Rules for the network Simplex Algorithm", *Networks* 20 (1990), 79-91.

[40] Goldfard D., Grigoriadis M. D., "A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow", *Annals of Operations Research* 13 (1988) 83-123.

[41] González-Martín, C., Metodos Interactivos en Programacion Multiobjetivo, Secretariado de Publicaciones de la Universidad de La Laguna, serie monografías 24 (1986).

[42] Grigoriadis, M. D., "An efficient implementation of the network simplex meted", Mathematical Programming Study 26 (1986) 83-111.

[43] Hu, T. C., "Multi-Commodity Network Flows", *Operations Research* 11 (1963) 344-360.

[44] Iri, M., "A new method of solving transportations-network problems", *Journal of the Operations Research Society of Japan* 2 (1960) 27-87.

[45] Isermann, H., "The enumeration of all efficient solutions for a linear multiple-objective Transportation problem", *Naval Research Logistics Quarterly* 26 (1979) 123-139.

[46] Jewell, W. S., "Optimal flow through networks", *interim Technical Report* 8, Operations Research Center, MIT, Cambridge, MA (1958).

[47] John, P. W., *Statistical Design and Analysis of Experiments*. Macmillan Company (1971).

[48] Johnson, E. L., "Networks and basic solutions", *Operations Research* 14 (1966) 619-624.

[49] Karzanov, A. V., "Determining the Maximal Flow in a Network by the method of Preflows", *Soviet Mathematical Doklady* 15 (1974) 434-437.

[50] Klein, M., “ A primal method for minimal cost flow flows with application to the assignment and transportation problems”, *Magnagement Science* 14 (1967), 205-220.

[51] Klingman, D. and Mote, J., “Solution approaches for network flow problems with multiple criteria”, *Advances in Management Studies* 1 (1982) (1), 1-30.

[52] Klingman, D., Napier, A. and Stutz, J., “NETGEN-a program for generating large scale (un) capacitated assignment, transportation and minimum cost flow network problems”, *Management Science* 20 (1974), 814-822.

[53] Lee, H., Pulat, S., “Bicriteria network flow problems: Continuous case”, *European Journal of Operational Research* 51 (1991), 119-126.

[54] Lee, H., Pulat, S., “Bicriteria network flow problems: Integer case”, *European Journal of Operational Research* 66 (1993), 148-157.

[55] Malhotra, R. and Puri, M., C., “Bi-criteria network problem”, *cahiers du C.E.R.O* 26 (1984), 95-102.

[56] Malhotra, V. M., Kumar, M. P, Maheshwari, S. N., “An O($|V|^3$) Algorithm for finding Maximum Flows in Networks”. *Inform. Process. Lett.* 7 (1978) 277-278.

[57] Milliken, G. A., Johnson, D. G., *Analysis of Messy Data, Volume 1: Designed Experiments*. Chapman and Hall (1992).

[58] Minty, G. J., “Monotone networks”, Proceedings of the Royal Society of London 257A (1960) 195-212.

[59] Mulvey, J., “Pivot strategies for primal-simplex network codes”, *Journal of ACM* 25 (1978) 266-270.

[60] Mustafa, A., Goh, M., “Finding integer efficient solutions for bicriteria and tricriteria network flow problems using Dinas”, *Computers and Operations Research* 25 (1998), 139-157.

[61] Nicoloso, S., Simeone, B., Classical and Contemporary Methods in Network *Optimization, part I: Network Flows* (1992).

[62] Pulat, P., Huarng, F. and Lee, H., “Efficient solutions for the bicriteria network flow problem”, *Computers and Operations Research*, 19 (7) (1992), 649-655.

[63] Ringuest, J. L. and Rinks, D. B., “Interactive solutions for the linear multiobjective Transportation problem”, *European Journal of Operational Research* 32 (1987) 96-106.

[64] Rothschild, B., Whinston, A., “On Two Commodity Network Flows”, *Operations Research* 14 (1966) 377-387.

[65] Sakarovitch, M., “Two Commodity Network Flows and Linear Programming”, *Mathematical Programming* 4 (1973) 1-20.

[66] *SAS/Stat, Procedure Anova*. Release 6.03 (1991).

[67] Sedeño-Noda A., González-Sierra, M.G., González-Martín, C., “An Algorithmic Study of the Maximum Flow Problem: A Comparative Statistical Analisis”. *TOP* 8 (1) (2000) 135-162.

[68] Sedeño-Noda, A., C. González-Martín, “An O($nm \log(U/n)$)-Time Maximum-

Flow Algorithm", *Naval Research Logistics Quarterly* 47 (6) (2000) 511-520.

[69] Sedeño-Noda, A., González-Martín, C. "Una variante del algoritmo de Ahuja-Orlin para problemas de Flujo Máximo: Experiencias computacionales y compraciones", Qüestiió 20 (3) (1996) 485-501.

[70] Sedeño-Noda, A., González-Martín, C., "An algorithm for the Biobjective Integer Minimum Cost Flow Problem", *Computers & Operations Research* 28 (2) (2001) 139-156.

[71] Sedeño-Noda, A., González-Martín, C., "The Biobjective minimum cost flow problem", *European Journal of Operational Research* 124 (2000) 591-600.

[72] Seymour, P. D., "A Two Commodity cut Theorem", *Discrete Mathematics* 23 (1978) 177-181.

[73] Sleator, D. D, Tarjan, R. E., "A Data Structure for Dynamic Trees". *Journal of Computer and System Sciences* 24 (1983) 362-391.

[74] Steuer, R. E., *Multiple-criteria optimization: theory, computation and application*, Wiley series in probability and mathematical statistics-applied (1985).

[75] Yu, P, *Multiple-criteria decision making*, Plenum Press (1985).

[76] Yu, P. and Zeleny, M., "Linear multiparametric programming by multicriteria simplex method", *Management Science* 23 (2) (1976), 159-170.