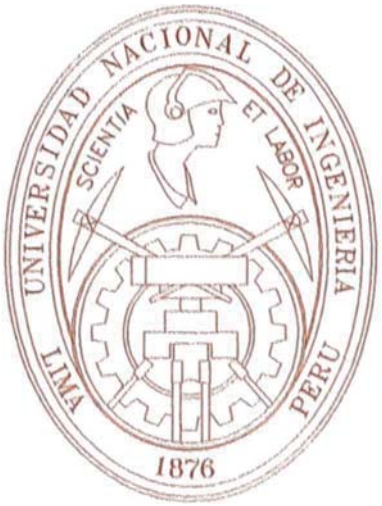
UNIVERSIDAD NACIONAL DE INGENIERIA
FACULTAD DE CIENCIAS
ESCUELA PROFESIONAL DE MATEMÁTICAS



ALGORITMOS POLINOMIALES PARA EL PROBLEMA DEL FLUJO MÁXIMO

TESIS

PARA OPTAR EL TÍTULO PROFESIONAL DE:
LICENCIATURA , ESPECIALIDAD : MATEMÁTICA

PRESENTADA POR :

RÓSULO PÉREZ CUPE

LIMA-PERU

2001

Dedicatoria

Este trabajo está dedicado muy especialmente a la memoria de mi madre Toribia. Asimismo con todo cariño a mi padre Doroteo, a mi hijo Paulo, mi esposa Liliana y finalmente a mis hermanos y sobrinos.

Agradecimientos

Aprovecho la oportunidad para agradecer al profesor Mg. William Echegaray por su apoyo y estímulo en la elaboración del presente trabajo, asimismo a los profesores Dr. Carlos Eduardo Ferreira, Dr. Paulo Feofiloff y Dra. Nami Kobayashi (U.S.P) por su motivación en el tema. De la misma manera a todos los profesores de la Facultad de Ciencias de la UNI que me apoyaron en todo momento. Y finalmente a todas aquellas personas que influenciaron en la elaboración del trabajo.

Índice

Resumen

En el presente trabajo se estudia el teorema del flujo máximo–corte mínimo (L. Ford y D. Fulkerson) desde el punto de vista práctico, esto es, su demostración se basa en la prueba de correctitud del algoritmo de etiquetas. Además se estudian diversas implementaciones de tal algoritmo con el fin de mejorar el tiempo de ejecución del mismo, utilizando para ello diferentes estrategias como son: caminos de capacidad relativamente grande; caminos con el menor número de arcos o Preflujos (Preflow Push) ésta última técnica es de reciente realización y es el que mejor funciona en la práctica.

Introducción

En este trabajo de tesis estudiamos ciertos algoritmos para la solución del problema del flujo máximo, este problema modela diversas situaciones reales. En particular resaltamos los algoritmos polinomiales, es decir aquellos que son prácticos, estudiamos conceptos modernos como son la red residual y también algoritmos recientes como son los algoritmos Preflow Push.

Para todos los algoritmos estudiamos su fundamento teórico, la implementación (casi en programa) y seguidamente un análisis matemático de su correctitud y complejidad de tiempo, esto último precisamente para justificar que el algoritmo sea polinomial, el contenido del trabajo está estructurado como sigue:

En el **Capítulo 1** elaboramos los elementos necesarios para el tratamiento posterior de los algoritmos que resuelven el problema. Básicamente aquí se ven conceptos de teoría de grafos, análisis de algoritmos y estructura de datos.

En el **Capítulo 2** estudiamos los algoritmos básicos para resolver el problema, en particular el algoritmo de etiquetas cuya complejidad no es polinomial, sin embargo es el punto de partida para la elaboración de otros algoritmos más eficientes y mas importante es el hecho de que su correctitud demuestra el teorema del flujo máximo–corte mínimo.

En el **Capítulo 3** se desarrollan los algoritmos polinomiales que son una sofisticación del algoritmo de etiquetas visto en el capítulo 2. Por ejemplo el algoritmo "Capacity Scaling" considera caminos de capacidad grande, mientras que el algoritmo del camino aumentante mas corto considera caminos con el menor número de arcos. Asimismo en este capítulo se estudia un tipo de algoritmo de reciente realización llamado algoritmo "Preflow push", estos envían flujos a lo largo de arcos a diferencia de los anteriores que envían flujo a lo largo de caminos.

En el **Capítulo 4** se presenta la parte práctica del trabajo como son algunas aplicaciones, programas y resultados numéricos.
En el **Capítulo 5** se presentan las conclusiones del trabajo a modo de resumen.

Capítulo 1

PRELIMINARES

A continuación daremos una serie de notaciones y definiciones que serán útiles a lo largo de todo el presente trabajo, éstos comprenden todos los conceptos necesarios en la teoría de grafos y redes así como su implementación. También se estudia el análisis de algoritmos mediante su correctitud y complejidad, utilizando para ello la notación asintótica de funciones como herramienta matemática. Finalmente se menciona la definición de algoritmo polinómico, así como la necesidad de construir algoritmos de este tipo en la solución de los diversos problemas a estudiar.

1.1 Introducción a la Teoría de Grafos

Cardinalidad Si A es un conjunto no vacío cualquiera, $|A|$ denota la cardinalidad o número de elementos del conjunto A.

Grafo Dirigido Un grafo dirigido (denotado por $G = (N, A)$), es un objeto matemático compuesto de nodos y arcos, donde N representa el conjunto de nodos y A representa el conjunto de arcos. Los nodos y arcos también serán llamados vértices y aristas del grafo respectivamente.

Los elementos de N son objetos cualesquiera, en el presente trabajo N tiene cardinalidad finita, usualmente $N = \{1, 2, 3, \ldots, n\}$ para algún número natural n.

Por otra parte A representa el conjunto de arcos, cuyos elementos son pares ordenados (i,j) de N ($i.e \quad A \subset (N \times N) \setminus \{(i,i)/i \in N\}$).

Generalmente en el presente trabajo usamos la notación $|N| = n$ y $|A| = m$, para denotar el número de nodos y arcos respectivamente. Por ejemplo en el grafo de la figura (1.1) tenemos el grafo dirigido $G = (N, A)$ donde $N = \{1,2,3,4,5,6\}$; $A = \{(1,2),(1,3),(2,3),(2,4),(3,5),(4,3),(4,6),(5,4),(5,6)\}$; $n = 6$ y $m = 9$.

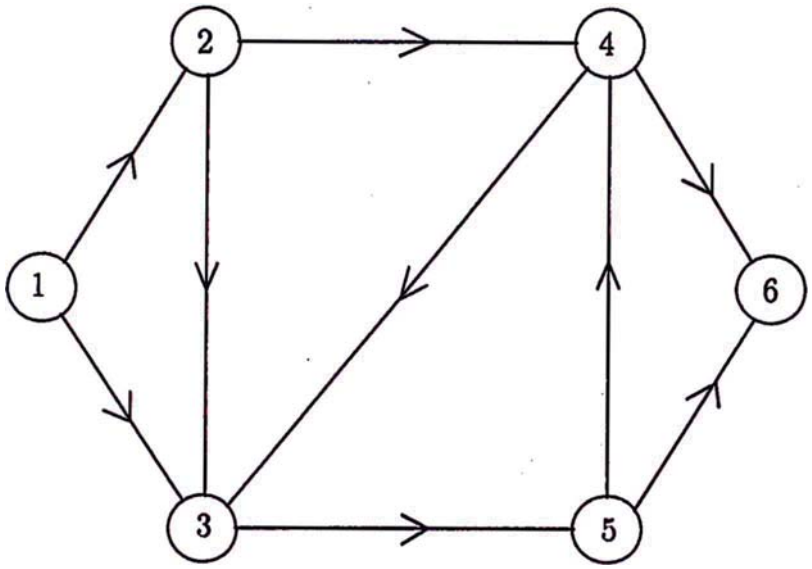


Figura 1.1: Grafo Dirigido G=(N,A)

Red Dirigida Una red dirigida es un grafo dirigido cuyos arcos y/o nodos tienen asociados números enteros, estos números típicamente representan costos, capacidades o flujos en el caso de arcos y ofertas o demandas en el caso de nodos; por ejemplo el grafo anterior se convierte en red (figura 1.2) si asociamos a cada arco un número que describe el costo de pasar por dicho arco; por ejemplo al arco (2,4) se asocia el número 9 que podría representar según sea el contexto el costo en el que se incurre al usar dicho arco.

Cola y Cabeza Sea el arco dirigido $(i,j) \in A$; i y j se llaman extremos del arco, al extremo i se le conoce como nodo inicial o cola del arco (i,j) y al extremo j se le conoce como nodo final o cabeza del arco (i,j); En ocasiones diremos que el arco (i,j) emana o sale desde el nodo i y termina o entra en el nodo j o en su defecto diremos que el nodo j es adyacente al nodo i.

Por ejemplo en la red de la figura (1.1) si tomamos el arco dirigido $(3,5)$ afirmamos que 3 es el nodo inicial y 5 el nodo final; tambien se dice que el nodo 5 es adyacente al nodo 3.

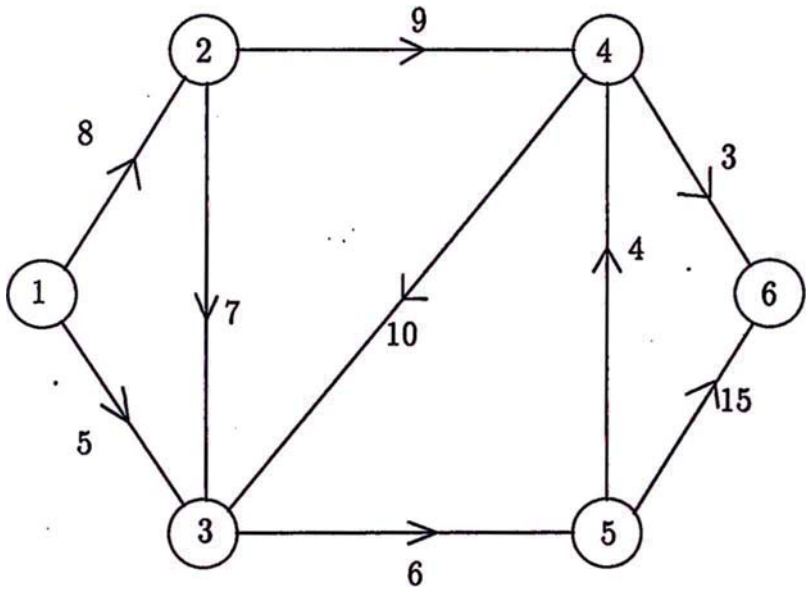


Figura 1.2: Red Dirigida G=(N,A)

Para las siguientes definiciones sea $G = (N, A)$ una red dirigida e $i \in N$ un nodo cualquiera.

Grado Interno El grado interno de i, denotado por $gi(i)$ es el número de arcos que entran a dicho nodo, es decir: $gi(i) = \left|\{j/(j,i) \in A\}\right|$.

Grado Externo El grado externo de i denotado por $ge(i)$ es el número de arcos que salen de dicho nodo, es decir: $ge(i) = \left|\{j/(i,j) \in A\}\right|$.

Grado El grado del nodo i, denotado por $g(i)$ es la suma del grado interno con el grado externo, es decir: $g(i) = gi(i) + ge(i)$. Por ejemplo en el grafo de la figura (1.1) $gi(5) = 1$; $ge(5) = 2$ y $g(5) = gi(5) + ge(5) = 1 + 2 = 3$.

También se tiene una propiedad interesante: $\sum_{i \in N} gi(i) = \sum_{i \in N} ge(i) = m$, por ejemplo en la figura (1.1) se tiene $gi(1) = 0$; $gi(2) = 1$; $gi(3) = 3$; $gi(4) = 2$; $gi(5) = 1$; $gi(6) = 2$, mientras que $ge(1) = 2$; $ge(2) = 2$; $ge(3) = 1$; $ge(4) = 2$; $ge(5) = 2$; $ge(6) = 0$; además se observa que: $\sum_{i=1}^{6} gi(i) = \sum_{i=1}^{6} ge(i) = 9$

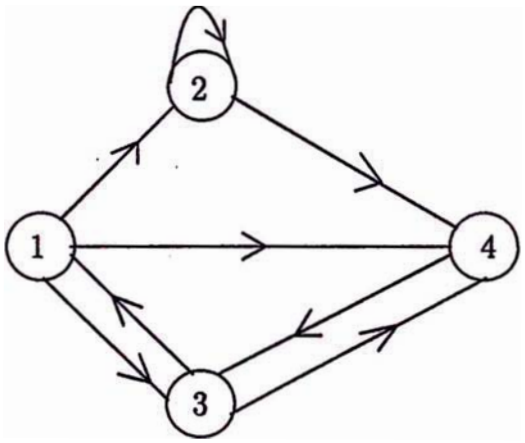


Figura 1.3: Grafo con arcos paralelos (3,4) y (4,3) y el lazo (2,2)

Lista de Adyacencia Sea $i \in N$ un nodo cualquiera, la lista de adyacencia del nodo i denotado por $A(i)$ se define indistintamente como:

- El conjunto de arcos que salen del nodo i

  $A(i) = \{(i,j) \in A \quad / \quad j \in N\}$ ó

- El conjunto de nodos adyacentes al nodo i

  $A(i) = \{j \in N \quad / \quad (i,j) \in A\}$

Por ejemplo en el grafo de la figura (1.3) $A(1) = \{2, 4, 3\}$ ó
$A(1) = \{(1,2), (1,4), (1,3)\}$

Arcos Paralelos y Lazos Dos o más arcos son paralelos cuando ellos tienen el mismo nodo inicial y el mismo nodo final; por ejemplo los arcos (3,4) y (4,3) de la figura (1.3) son paralelos, por otra parte se dice que un arco es un lazo si el nodo inicial es igual al nodo final; un lazo es un arco de la forma (i, i) con $i \in N$. Por ejemplo el arco (2,2) de la figura (1.3) es un lazo.

Subgrafo Un Grafo $G' = (N', A')$ es un subgrafo de $G = (N, A)$ si $N' \subseteq N$ y $A' \subset A$ y todo arco en G' es también arco en G. Por ejemplo si consideramos $N' = \{1, 2, 4\}$ y $A' = \{(2,2), (1,4), (2,4)\}$ tenemos el subgrafo $G' = (N', A')$ (figura 1.4) del grafo $G = (N, A)$ de la figura (1.3).

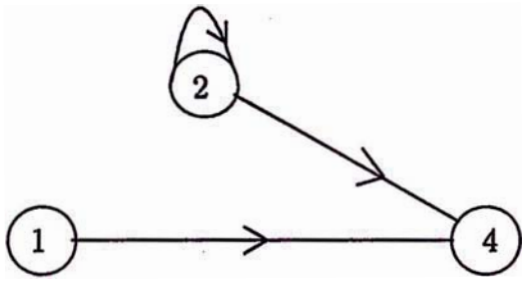


Figura 1.4: Subgrafo de la figura (1.3)

Paseo Dirigido Un Paseo Dirigido en un grafo G es un subgrafo que consiste de una secuencia arbitraria de nodos y arcos $i_1 - a_1 - i_2 - a_2 - \ldots - i_{r-1} - a_{r-1} - i_r$ donde se tiene que $a_k = (i_k, i_{k+1}) \in A \; \forall \quad 1 \leq k \leq r - 1$. Dicho en palabras es una secuencia arbitraria de nodos y arcos donde se respeta la dirección de los arcos. Generalmente un paseo dirigido se denota como una secuencia de nodos sin mención explícita de arcos. Por ejemplo en el grafo de la figura (1.3), la secuencia de nodos y arcos: 1-(1,2)-2-(2,2)-2-(2,4)-4-(4,3)-3-(3,1)-1-(1,4)-4; es un paseo dirigido que respeta las

Implementación En los programas que implementaremos más adelante los grafos serán representados como lista de adyacencia, la cual es un vector (de longitud igual a la cantidad de nodos) que guarda direcciones que a su vez son listas ligadas; cada nodo de dicha lista guarda el adyacente al nodo actual y un apuntador al próximo de la lista, si es necesario también se puede guardar la información sobre el arco (caso de una red dirigida). Por ejemplo para la red de la figura (1.2) su representación como lista de adyacencia es como se muestra en la figura siguiente:

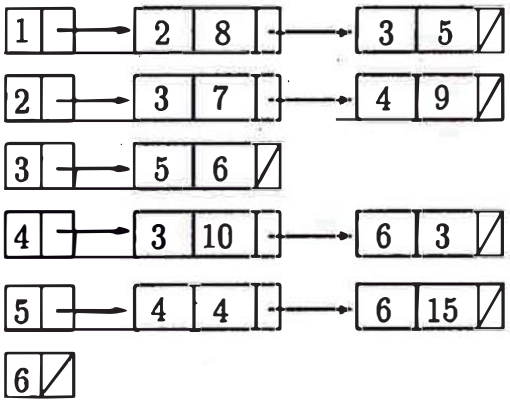


Figura 1.6: Grafo representado como lista de adyacencia (vector de listas teniendo como índice los nodos del grafo). La primera lista representa al arco (1,2) que tiene asociado el número 8, a continuación el arco (1,3) que tiene asociado el número 5; la lista 3 tiene un solo vértice adyacente mientras que la lista 6 no tiene vértices adyacentes

1.2 Análisis de Algoritmos

En esta sección estudiaremos el método para **analizar** un algoritmo, entendemos por **analizar** un algoritmo estudiar la **correctitud y complejidad** del mismo, conceptos que definiremos a continuación.

Problema En nuestro contexto, un problema es la interpretación de alguna situación de la vida real mediante un modelo matemático, la misma que requiere solución por algún método algorítmico, es por esta razón que los problemas que estudiamos se denominan **problemas computacionales**. Estos problemas se caracterizan por tener un conjunto de parámetros y un enunciado sobre las propiedades que la respuesta o solución del problema debe satisfacer.

Instancia Una instancia de un problema se obtiene al fijar valores particulares a cada uno de los parámetros.

Algoritmo Es una herramienta para resolver problemas computacionales, se compone de un conjunto de pasos finitas y bien definidos. El algoritmo recibe un conjunto de datos (**entrada**) y produce otro conjunto de datos (**salida**) mediante la aplicación del conjunto de pasos mencionado.

Los algoritmos en este trabajo son descritos mediante un conjunto reducido de palabras reservadas (escritas en negrilla), que son muy parecidos a los usados en la mayoría de los lenguages de programación, los que usamos son los siguientes:

mientras $\cdots$ **hacer** ;

repetir $\cdots$ **hasta** ;

para $\cdots$ **hasta** $\cdots$ **hacer** y

si $\cdots$ **entonces** $\cdots$ **sino** .

Ejemplo 1.1 *Para ilustrar las definiciones anteriores, consideremos la siguiente situación real: La Universidad Nacional de Ingeniería tiene cierta cantidad de alumnos en el nivel de pregrado, estamos interesados en generar un reporte de alumnos en estricto orden de mérito con respecto a las notas obtenidas por dichos alumnos en los cursos comunes (cursos básicos) a todas las facultades.*

El **problema** *o modelo matemático para dicha situación sería el llamado problema de ordenación de una secuencia de números; es decir dados n números reales* $a_1, a_2, \cdots, a_n$, *obtener una permutación de los mismos dado por* $a'_1, a'_2, \cdots, a'_n$ *tal que* $a'_1 \leq a'_2 \leq \cdots \leq a'_n$, *los parámetros para este problema son n y los* $a_i \quad i = 1, 2, \cdots, n$

Una **instancia** *para este problema podría ser por ejemplo* $n = 6$ *y* $a_1 = 10.3$, $a_2 = 5.6$, $a_3 = 12.7$, $a_4 = 9.5$, $a_5 = 12.8$ *y* $a_6 = 10.1$

Un **algoritmo** *para resolver el problema anterior puede ser el algoritmo de ordenación por inserción (existen muchos otros), este algoritmo es mostrado a continuación:*

Algoritmo de Inserción

Entrada: $n, A_1, A_2, \cdots, A_n$

Salida: $A'_1, A'_2, \cdots, A'_n$ *(tal que* $A'_1 \leq A'_2 \leq \cdots \leq A'_n$*)*

```
1   j := 2;
2   mientras j ≤ n hacer:
3       aux := A[j]
4       i := j − 1;
5       mientras (i > 0) y (A[i] > aux) hacer:
6           A[i + 1] := A[i];
7           i := i − 1;
8       A[i + 1] := aux
9       j := j + 1;
10  devolver(A)
```

Figura 1.7: Algoritmo de Inserción

1.2.1 Correctitud

Decimos que un algoritmo es correcto, si para cada instancia de entrada el algoritmo termina con la respuesta esperada. La prueba formal de la correctitud de los algoritmos no es fácil, debido a que cada algoritmo tiene sus propias caracteristicas. Comúnmente se utiliza el método de los **invariantes** asi como el de la **inducción matemática.**

En el caso específico del algoritmo anterior (figura 1.7) sean dados n y la secuencia de reales $\langle A[1], A[2], \cdots, A[n] \rangle$ demostramos por inducción que la secuencia $\langle A[1], A[2], \cdots, A[j-1] \rangle$ está ordenada para todo j (ésta es la invariante para el algoritmo en cada iteración).

La base de la inducción es para $j = 2$ (secuencia $\langle A[1], \cdots, A[2-1]\rangle = A[1]$) el cual evidentemente está ordenado.

La hipótesis de inducción será que la secuencia $\langle A[1], A[2] \cdots, A[j-1]\rangle$ está ordenada y veamos qué sucede con la secuencia $\langle A[1], A[2] \cdots, A[j]\rangle$ (es decir para j); el loop 5-6-7-5 (figura 1.7) termina cuando la condición de la linea 5 es falsa, esto puede suceder solo en dos casos:

- Cuando $i = 0$, es decir cuando la secuencia está totalmente deslocada, en este caso $A[i] > aux \ \forall\, i = 1..j-1$ y luego de ejecutarse las lineas 8 y 9 tenemos la secuencia de j elementos $\langle aux, A[1], A[2], \cdots, A[j-1]\rangle$ la cual está ordenada. O cuando,
- $A[i] \leq aux$ y $A[\ell] > aux$ para $\ell = i+1, i+2, \cdots, j-1$ y luego de ejecutarse las lineas 8 y 9 tenemos la secuencia de j elementos $\langle A[1], \cdots, A[i], aux, A[i+1], \cdots A[j-1]\rangle$, la misma que por las condiciones también está ordenada.

Luego entonces la invariante se cumple para todas las iteraciones, en particular cuando $j = n + 1$ en cuyo caso la secuencia $\langle A[1], A[2], \cdots, A[n]\rangle$ está ordenada, por lo tanto al terminar el algoritmo retorna el vector A cuyos elementos están ordenados, esto prueba la correctitud del algoritmo.

1.2.2 Complejidad

Analizar la complejidad de un algoritmo consiste en determinar cuántos recursos necesita el mismo; éstos pueden ser memoria o tiempo de ejecución, en este trabajo y casi generalmente se considera como complejidad el tiempo. Este tiempo normalmente está en función de los parámetros de entrada (llamado también tamaño del problema), dicho tamaño depende de cada problema, por ejemplo para el problema de ordenación el tamaño natural es el número de datos a ser ordenado; por otro lado para analizar un algoritmo de multiplicación de dos enteros una medida natural podría ser el número total de bits necesarios para almacenar cada entero. Si la entrada de un algoritmo es un grafo es más útil describir el tamaño del mismo mediante el número de arcos y nodos.

Para éste analisis asumimos que las instrucciones son ejecutadas secuencialmente una después de la otra (modelo RAM) , generalmente no es posible obtener una expresión

matemática exacta para la complejidad de tiempo del algoritmo, conformándonos tan solamente con una cota superior para dicha complejidad, la cual se denominará **complejidad en el peor caso**, para lo cual utilizaremos la notación asintótica que se describe mas adelante (notación $\mathcal{O}$).

Veamos entonces el análisis de complejidad (tiempo) del algoritmo de inserción mostrado en la figura (1.7), para realizar un análisis independiente de la máquina utilizada asumimos que cada instrucción del algoritmo anterior requiere de un tiempo genérico. Por ejemplo la linea 1 demora un tiempo c_1; la linea 2 un tiempo c_2; y así sucesivamente la linea 9 un tiempo c_9, siendo cada uno de los c_i constantes.

El tiempo total utilizado por una determinada linea de instrucción está dado por el producto del número de veces que dicha linea es ejecutada por el tiempo que requiere la misma (c_i $i = 1, \cdots, 9$). Luego la parte central del análisis y por tanto la que requiere mas trabajo es determinar cuántas veces se ejecuta cada linea, esto es mostrado en la siguiente tabla:

Linea de programa	Tiempo genérico	Número de veces que se ejecuta la linea
1	c_1	1
2	c_2	n
3	c_3	$n-1$
4	c_4	$n-1$
5	c_5	$\sum_{j=2}^{n} t_j$
6	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	c_8	$n-1$
9	c_9	$n-1$

Tabla 1.1: Cálculo del número de veces que cada lines es ejecutado

Observaciones:

- Para cada j de la linea 2 el loop 5-6-7-5 del algoritmo de la figura (1.7) se ejecuta un cierto número de veces, sea dicho número t_j, luego las lineas 6 y 7 se ejecutan $(t_j - 1)$ veces para cada j y durante la ejecución de todo el algoritmo dichas lineas se ejecutan $\sum_{j=2}^{n}(t_j - 1)$ veces como se muestra en la tabla anterior.
- $1 \leq t_j \leq j \ \forall \ j = 2, \cdots, n$; si $t_j = 1 \ \forall \ j$, entonces la secuencia está ordenada; por otro lado si $t_j = j \ \forall \ j$, entonces la secuencia está ordenada a la inversa (de mayor a menor).

Finalmente la complejidad de tiempo del algoritmo está dada por la expresión:

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) + c_9(n-1) \qquad (1.1)$$

La expresión anterior puede ser simplificada si consideramos que $t_j = 1$ para todo j entonces tenemos la llamada **complejidad mas favorable** dada por:

$$\begin{aligned} T(n) &= c_1 + c_2 n + (c_3 + c_4 + c_5 + c_8 + c_9)(n-1) \\ &= c_1 - (c_3 + c_4 + c_5 + c_8 + c_9) + (c_2 + c_3 + c_4 + c_5 + c_8 + c_9)n \\ &= A + Bn \end{aligned} \qquad (1.2)$$

Siendo A y B constantes.

Por otro lado si consideramos $t_j = j$ para todo j obtenemos la llamada **complejidad mas desfavorable** dado por:

$$\begin{aligned} T(n) &= c_1 + c_2 n + (c_3 + c_4 + c_8 + c_9)(n-1) + c_5 \frac{(n+2)(n-1)}{2} + (c_6 + c_7)\frac{(n)(n-1)}{2} \\ &= c_1 - (c_3 + c_4 + c_8 + c_9) - c_5 + (c_2 + c_3 + c_4 + c_8 + c_9 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2})n + \\ &\quad + (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 \\ &= An^2 + Bn + C \end{aligned} \qquad (1.3)$$

donde A,B y C son constantes.

En el presente trabajo utilizaremos siempre la complejidad de tiempo en el peor caso es decir la situación mas desfavorable. Además despreciamos constantes y términos de menor grado, utilizando para ello la notación asintótica que se describe a continuación.

1.3 Notación asintótica ($\mathcal{O}$)

Definición 1.3.1 (Notación $\mathcal{O}$)

Si n es el tamaño del problema y f, g representan funciones de n, es decir $f, g : \mathbb{N} \longrightarrow \mathbb{R}$, entonces:

$$\mathcal{O}(g(n)) = \{f(n) \ / \ \exists c \in \mathbb{R}, \quad \exists n_0 \in \mathbb{N} \ tal \ que \ f(n) \leq c \cdot g(n) \quad \forall n \geq n_0\} \qquad (1.4)$$

Por abuso de notación y para efectos prácticos escribimos $f(n) = \mathcal{O}(g(n))$, donde $f(n)$ representa la complejidad del algoritmo y $c \cdot g(n)$ es la cota superior al cual hacíamos alusión en la sección anterior, por ejemplo si la complejidad de tiempo de un algoritmo está dado por $f(n) = \mathcal{O}(n^2)$ entonces afirmamos informalmente que el tiempo $f(n)$ que le toma al algoritmo resolver el problema es menor o igual a $c \cdot n^2$ para alguna constante c.

Sin embargo formalmente debe existir $n_0 \in \mathbb{N}$ y $c \in \mathbb{R}$ tal que la complejidad del algoritmo $f(n)$ no supere el valor de $c \cdot g(n)$ para todo $n \geq n_0$, a pesar de que para $n < n_0$ la situación es impredecible. En situaciones prácticas el valor de n_0 es relativamente grande, debido a que se quiere conocer el comportamiento de los algoritmos para entradas de tamaño grande, gráficamente:

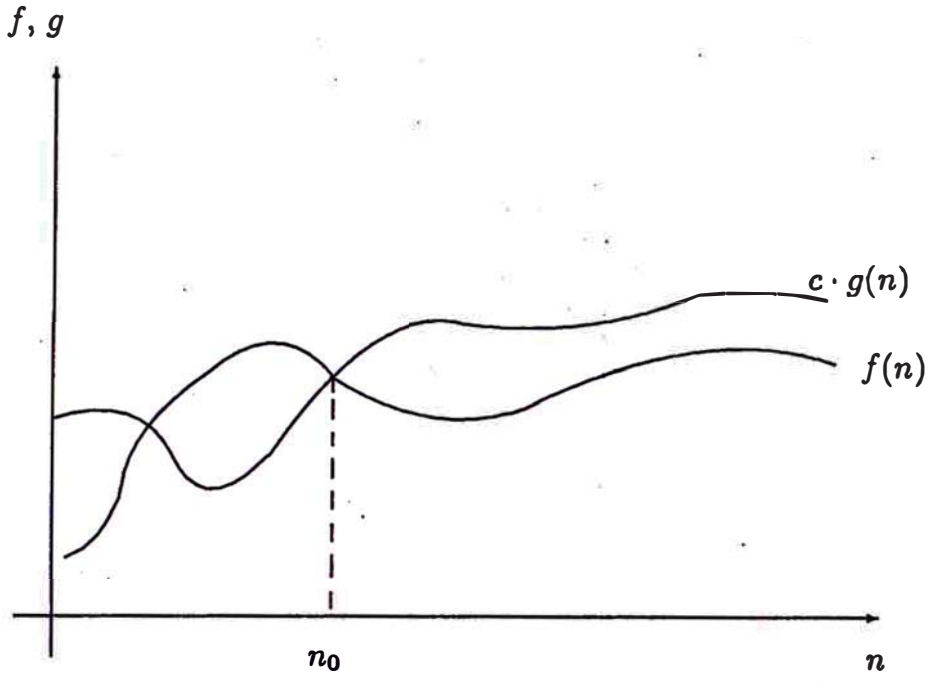


Figura 1.8: Notación $\mathcal{O}$

Afirmar que un algoritmo tiene complejidad $g(n)$ es equivalente a escribir $f(n) = \mathcal{O}(g(n))$; dependiendo de la función $g(n)$ podemos clasificar los algoritmos en la forma siguiente:

Función $g(n)$	Complejidad ($f(n)$)	Denominación del algoritmo
1 (o cuaquier cte $c > 0$)	$\mathcal{O}(1)$	constante
n	$\mathcal{O}(n)$	lineal
n^2	$\mathcal{O}(n^2)$	cuadrática
n^3	$\mathcal{O}(n^3)$	cúbica
n^k ($k \in \mathbb{N}$)	$\mathcal{O}(n^k)$	polinomial
$Log(n)$	$\mathcal{O}(Log(n))$	logarítmica
$nLog(n)$	$\mathcal{O}(nLog(n))$	polinomial
c^n ($c > 1$: cte)	$\mathcal{O}(c^n)$	exponencial

Tabla 1.2: Clasificación de algoritmos

A partir de la definición tenemos la siguiente propiedad:

Propiedad 1.3.1 *Si f , g y h son funciones de $\mathbb{N}$ a $\mathbb{R}$ y k es cualquier constante real, entonces:*

1. $\mathcal{O}(kf(n)) = \mathcal{O}(f(n))$
2. $\mathcal{O}(f(n) + k) = \mathcal{O}(f(n))$
3. $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n)) + \mathcal{O}(g(n))$
4. $\mathcal{O}(f(n)g(n)) = \mathcal{O}(f(n))\mathcal{O}(g(n))$
5. $f(n) = \mathcal{O}(f(n))$ *(reflexiva)*
6. *Si* $f(n) = \mathcal{O}(g(n))$ *y* $g(n) = \mathcal{O}(h(n))$ *entonces* $f(n) = \mathcal{O}(h(n))$ *(transitiva)*

En consecuencia, al calcular la complejidad de un algoritmo podemos despreciar constantes aditivas o multiplicativas, por ejemplo si para un cierto algoritmo tenemos calculado que éste realiza $3n^2$ operaciones, entonces para efectos prácticos afirmamos que su complejidad es $T(n) = \mathcal{O}(n^2)$ o mas aún si otro algoritmo realiza $(6n^3 + 4n - 5)$ operaciones, entonces consideramos que su complejidad es $T(n) = \mathcal{O}(n^3)$.

Para terminar de analizar la complejidad del algoritmo de inserción descrito anteriormente (figura 1.7), el exfuerzo que éste realiza en el mejor de los casos según la ecuación (1.2) está dado por $T(n) = \mathcal{O}(n)$ operaciones y en el peor de los casos (según la ecuación 1.3) dicho exfuerzo está dado por $T(n) = \mathcal{O}(n^3)$ operaciones. Además debemos mencionar que existen otros algoritmos mas eficientes para el problema de ordenación cuyas complejidades son logaritmicas.

1.4 Algoritmos Polinomiales

Los algoritmos que tienen nuestra preferencia son aquellos que tienen un crecimiento asintótico moderado, como son los algoritmos polinomiales o aquellos logarítmicos, sin embargo esta afirmación tiene sus límites pues si bien es cierto que un algoritmo que realiza n^3 operaciones es bueno, sin embargo un algoritmo que realiza n^{10} (o mas aún n^{50})

operaciones ya no es práctico a pesar de que sería un algoritmo polinómico (en todo caso su estudio sólo tiene un sentido teórico).

Para reafirmar lo dicho anteriormente consideremos diversas funciones de complejidad ($T(n)$) y diversos tamaños de entrada (n); además consideramos que disponemos de una máquina que realiza 1 millón de operaciones por segundo (es decir la velocidad de la máquina está dado por $v = 10^6 \frac{operaciones}{seg}$)

Si $T(n)$ es el número de operaciones que realiza la máquina para ejecutar un determinado algoritmo y v es la velocidad de dicha máquina entonces el tiempo (en segundos) que le toma para ejecutar el algoritmo está dado por

$$t = \frac{T(n)}{v}$$

Los resultados para diversos algoritmos y diversos tamaños de entrada están dados en la siguiente tabla (1.3); donde claramente se observa que los algoritmos polinomiales son mejores que los exponenciales; sin embargo dentro de los polinomiales los más prácticos son los de grado pequeño.

	10	20	40	50	100	500
n	0.00001 s	0.00002 s	0.00004 s	0.00005 s	0.0001 s	0.0005 s
$nLog(n)$	0.00003 s	0.00008 s	0.00021 s	0.00028 s	0.00066 s	0.00448 s
n^2	0.0001 s	0.0004 s	0.0016 s	0.0025 s	0.01 s	0.25 s
n^3	0.001 s	0.008 s	0.64 s	0.125 s	1.00 s	125 s
n^5	0.1 s	3.2 s	1.7 min	5.2 min	166.6 min	1 año
n^{10}	2.7 horas	118 días	3.3 siglos	30 siglos	$3{\cdot}10^4 siglos$	$3{\cdot}10^{11} siglos$
2^n	0.001 s	1 s	12.7 dias	35.7 años	$4 \cdot 10^{14}$ siglos	10^{135} siglos
3^n	0.059 s	58 min	3853 siglos	10^8 siglos	10^{32} siglos	10^{223} siglos

Tabla 1.3: Comparación del tiempo para diferentes algoritmos

Además surge una pregunta natural ¿qué tan rápido debe ser una máquina para resolver en un tiempo razonable los algoritmos exponenciales? la respuesta a ésta pregunta es dada mediante el contenido de la tabla (1.4), la cual es analizada inmediatamente.

Supongamos que un computador actual con cierta velocidad resuelve en 1 hora problemas de tamaño T_i (segunda columna de la tabla 1.4) para 4 algoritmos diferentes cuyas complejidades son dadas en la primera columna de dicha tabla. En la tercera columna tenemos los tamaños de los mismos problemas pero resueltos con un computador 100 veces mas rápido en el mismo tiempo (1 hora) y similarmente en la cuarta columna con un computador 1000 veces mas rápido.

funcion de complejidad	Computador actual	100 veces mas rápido	1000 veces mas rápido
n	T_1	$100T_1$	$1000T_1$
n^2	T_2	$10T_2$	$31.6T_2$
n^3	T_3	$4.6T_3$	$10T_3$
2^n	T_4	$T_4 + 6.6$	$T_4 + 10$

Tabla 1.4: Comparación de tamaño de problemas

Según el contenido de esta tabla, podemos afirmar que por ejemplo, para un algoritmo cuya complejidad es n^2 en 1 hora un computador 100 veces mas rápido es capáz de resolver un problema cuyo tamaño sea 10 veces mayor; y un computador 1000 veces mas rápido un problema de tamaño 32 veces mayor.

Sin embargo el algoritmo de complejidad exponencial no tiene muchas esperanzas; pues como se observa en la tabla, con un aumento de 100 veces (o 10000%) en velocidad, se prodría resolver un problema cuyo tamaño es 6 unidades más. Y si la velocidad es mejorada mas aún (1000 veces o en 100000%) el tamaño del problema a resolver sólo podría aumentar en 10 unidades.

Capítulo 2

ALGORITMOS BÁSICOS

En este capítulo estudiamos el método clásico de Ford–Fulkerson para resolver el problema en estudio. Llamamos **método** antes de llamar **algoritmo**, pues como todo método soporta varias implementaciones, cada una de las cuales tiene diferentes tiempos de ejecución, en este capítulo estudiaremos el algoritmo de **etiquetas** el cual no es un algoritmo polinomial, dejando para el siguiente capítulo los algoritmos **capacity scaling, shortest augmenting path** y **preflow push**, los cuales según veremos mas adelante si son polinomiales.

El método de Ford–Fulkerson ([7], [8] y [9]) utiliza tres conceptos importantes, que no solo son útiles para este método sino para muchos otros métodos y mas aún para otros problemas, estos conceptos son **red residual,camino aumentante** y **cortes** los mismos que serán presentados con detalle a continuación, asimismo estas ideas son claves para la comprensión y demostración del teorema del **flujo máximo–corte mínimo**

2.1 Notaciones y Definiciones

Definición 2.1.1 *Dado $G = (N, A)$ una red dirigida, definimos en general una función f sobre el conjunto de los arcos del siguiente modo:*

$$\begin{aligned} f: \quad A &\longrightarrow \mathbb{Z}^+ \\ (i,j) &\longrightarrow f(i,j) := f_{ij} \end{aligned} \tag{2.1}$$

Donde $\mathbb{Z}^+ := \{0, 1, 2, 3, \cdots\}$ enteros no negativos.

Generalmente la función f representa **capacidad** (u), **costo** (c) o **flujo** (x)

- si $f \equiv u \Longrightarrow u_{ij}$ representa la CAPACIDAD del arco (i,j)
- si $f \equiv c \Longrightarrow c_{ij}$ representa el COSTO al pasar por el arco (i,j)
- si $f \equiv x \Longrightarrow x_{ij}$ representa el FLUJO (a ser definido mas adelante) que pasa a través del arco (i,j)

Definición 2.1.2 *Sea $G = (N, A)$ una red dirigida y consideremos ahora un subconjunto S cualquiera de N $(S \subset N)$. Además sea $\overline{S} = (N - S)$ entonces definimos el conjunto:*

$$(S, N-S) = (S, \overline{S}) := \left\{ (i,j) \in A \Big/ \quad i \in S \quad , \quad j \in \overline{S} \right\} \tag{2.2}$$

El cual representa el conjunto de arcos que salen o nacen en S y entran o llegan a $\bar{S} = (N - S)$.

Definición 2.1.3 *Sea $G = (N, A)$ una red dirigida y sea $f : A \longrightarrow \mathbb{Z}^+$ cualquier función como el definido en la ecuación (2.1), entonces definimos:*

$$f(S, N-S) = f(S, \overline{S}) = \sum_{i \in S} \sum_{j \in \overline{S}} f_{ij} := \sum_{(i,j) \in (S, \overline{S})} f_{ij} \tag{2.3}$$

Esta cantidad representa la suma de los valores sobre los arcos considerados en (2.2). Este número puede ser visto como el valor total que sale de S y entra en $\overline{S} = (N - S)$

Para simplificar las operaciones posteriores adoptaremos la siguiente notación:

$$f(S) := f(S, \overline{S}) = f(S, N - S)$$

A modo de ilustrar las definiciones anteriores consideremos la red de la figura (2.1), donde $N = \{1, 2, 3, 4, 5, 6\}$ y la función $f = u$ (capacidad) definida sobre A

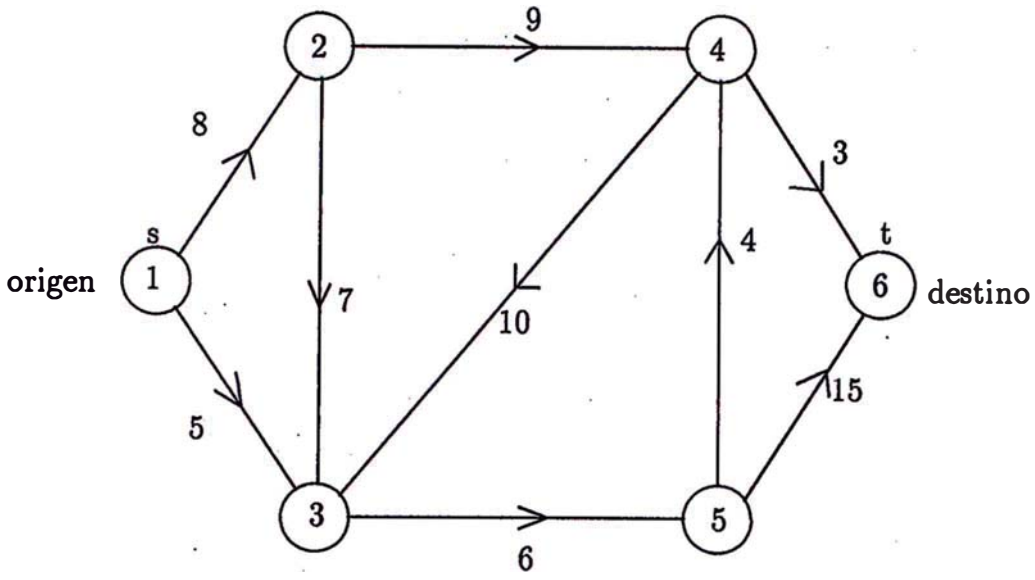


Figura 2.1: Red Dirigida $G = (N, A)$

Ejemplo 2.1 *Si consideramos:* $S = \{1,2,3\}$ $\overline{S} = (N - S) = \{4,5,6\}$
$\Longrightarrow f(S,\overline{S}) = u(S,\overline{S}) = u(S) = u_{24} + u_{35} = 9 + 6 = 15$

Ejemplo 2.2 *Por otra parte si ahora consideramos:* $S = \{1,3,4\}$ $\overline{S} = (N - S) = \{2,5,6\}$
$\Longrightarrow f(S,\overline{S}) = u(S,\overline{S}) = u(S) = u_{12} + u_{35} + u_{46} = 8 + 6 + 3 = 17$

Ejemplo 2.3 *Sea* $S = \{4\}$ $\overline{S} = (N - S) = \{1,2,3,5,6\}$
$\Longrightarrow f(S,\overline{S}) = u(S,\overline{S}) = u(S) = u(\{4\}) = u_{43} + u_{46} = 10 + 3 = 13$

Ejemplo 2.4 *Si* $S = \{4\}$ $\overline{S} = (N - S) = \{1,2,3,5,6\}$
$\Longrightarrow f(\overline{S},S) = u(\overline{S},S) = u(\overline{S}) = u_{24} + u_{54} = 9 + 4 = 13$

2.2 Formulación del Problema

Sea la red dirigida $G = (N, A)$ tomemos dos nodos arbitrarios diferentes; estos nodos serán tratados de un modo especial: uno de ellos será denominado **nodo origen** y denotado mediante $\underline{s}$; mientras que el otro será denominado **nodo destino** y denotado mediante $\underline{t}$ $(s \neq t)$.

Definición 2.2.1 (FLUJO)

Dados una red $G = (N, A)$ dos nodos s y t (llamados origen y destino respectivamente) un **flujo** *de s a t en G es una función:*

$$\begin{aligned} x: \quad A &\longrightarrow \mathbb{Z}^+ \\ (i,j) &\longrightarrow x(i,j) := x_{ij} \end{aligned} \tag{2.4}$$

Tal que:

$$x(\{i\}) = x(N - \{i\}) \quad \forall \quad i \in N \setminus \{s,t\} \tag{2.5}$$

La relación (2.5) es conocida como **condición de equilibrio de masa** el cual según (2.3) significa que el valor total que sale del nodo i ($x(\{i\})$) es igual al valor total que entra al nodo i ($x\{N - \{i\}\}$) para todo nodo i diferente de s y t.

Definición 2.2.2 *Sean $G = (N, A)$ una red; x un flujo de s a t en G, entonces el* **valor del flujo** *$v(x)$ es definido mediante:*

$$v(x) := x(\{s\}) - x(N - \{s\}) = -(x(\{t\}) - x(N - \{t\})) \tag{2.6}$$

El cual es interpretado como la cantidad de flujo que sale de s menos la cantidad de flujo que entra a s ó el negativo de lo que sale de t menos lo que entra en t

Observaciones:

1. Se utilizará $v(x)$ sólo si hay confusión o si se desea mencionar el flujo explícitamente; en otro caso solo se mencionará v.

2. Es claro que dados una red $G = (N, A)$, dos nodos especiales s y t (origen y destino respectivamente) existe una cantidad finita de flujos de s a t en G; cada uno con un determinado valor, luego tiene sentido hablar de aquel flujo cuyo valor es máximo.

Consideremos entonces una función :

$$\begin{aligned} u: \quad A &\longrightarrow \mathbb{Z}^+ \\ (i,j) &\longrightarrow u(i,j) := u_{ij} \end{aligned} \tag{2.7}$$

la cual se denominará función **capacidad**; por otro lado u_{ij} representa la máxima capacidad que tiene el arco (i, j), esta función generalmente es dato en los diversos algoritmos y mas aún en aplicaciones reales.

Definición 2.2.3 *Sean las funciones x (flujo) y u (capacidad) sobre una red $G = (N, A)$, se dice que un flujo x* **respeta** *u si:*

$$x_{ij} \leq u_{ij} \quad \forall \quad (i,j) \in A \tag{2.8}$$

Definición 2.2.4 (Flujo Máximo)
Sean $G = (N, A)$ una red capacitada con capacidad u, s y t dos nodos especiales (origen y destino) y $x^0 : A \longrightarrow \mathbb{Z}^+$ un flujo de s a t en G. Se dice que x^0 es un **flujo máximo** *si* **respeta** *u y su valor es máximo considerando todos los flujos de s a t en G que* **respetan** *u, es decir:*

$$v(x^0) = \max\{v(x) / \quad x \text{ es un flujo de } s \text{ a } t \text{ en } G \text{ que respeta } u\}$$

Ejemplo 2.5 *Consideremos el flujo x sobre la red de la figura (2.1), además sean $s = 1$ (nodo origen) y $t = 6$ (nodo destino), tal como se muestra en la siguiente figura:*

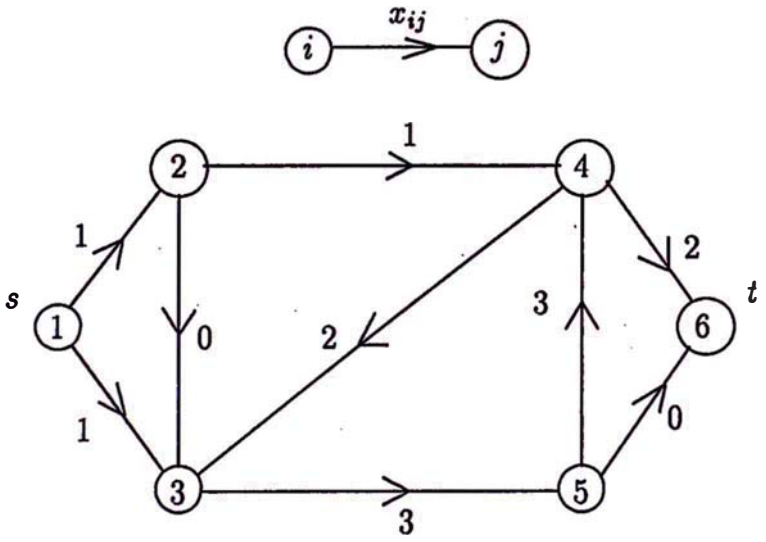


Figura 2.2: flujo x de $s = 1$ a $t = 6$ que **respeta** u y cuyo valor es $v(x) = 2$, el número sobre cada arco indica el flujo

Ejemplo 2.6 *Sobre la misma red de la figura (2.1), consideramos ahora otro flujo x con $s = 1$ y $t = 6$ (origen y destino respectivamente)*

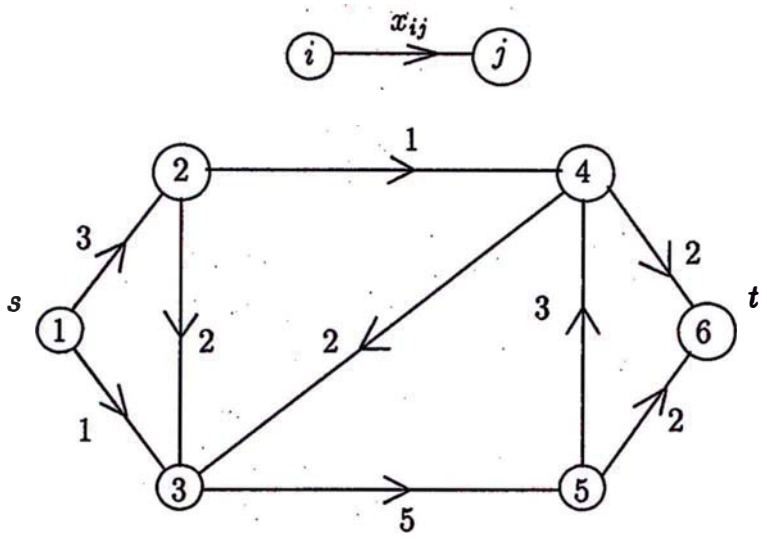


Figura 2.3: flujo de $s = 1$ a $t = 6$ que **respeta** u y cuyo valor es $v(x) = 4$

Ejemplo 2.7 *$x = 0$ también es un flujo de $s = 1$ a $t = 6$, cuyo valor es $v(0) = 0$*

Es claro que dada una red $G = (N, A)$ y una función capacidad $u : A \longrightarrow \mathbb{Z}^+$ existe un número finito de flujos x que **respetan** u (es decir $x_{ij} \leq u_{ij}$) y cada uno de ellos tiene un cierto valor. Con estos elementos estamos en condiciones de formular el problema que estudiaremos de aquí en adelante.

Definición 2.2.5 (Problema del Flujo Máximo)
Dados una red $G = (N, A)$; una función **capacidad** $u : A \longrightarrow \mathbb{Z}^+$; *dos nodos s y t (origen y destino respectivamente). Hallar un* **flujo** $x : A \longrightarrow \mathbb{Z}^+$ *de s a t que* **respete** u $(x_{ij} \leq u_{ij})$ *y cuyo valor sea máximo.*
Formalmente:

$$\max v(x)$$

s.a.

$$\sum_{\{j/(i,j)\in A\}} x_{ij} - \sum_{\{j/(j,i)\in A\}} x_{ji} = \begin{cases} v(x) & ; \ si \ \ i = s \\ 0 & ; \ si \ \ i \in N - \{s,t\} \\ -v(x) & ; \ si \ \ i = t \end{cases} \tag{2.9}$$

$$0 \leq x_{ij} \leq u_{ij} \qquad \forall\, (i,j) \in A \tag{2.10}$$

En la siguiente sección definiremos algunos conceptos para proceder a formular los algoritmos para resolver el problema planteado.

2.3 Cortes y Red Residual

Recordemos que dado una red $G = (N, A)$ un **corte** es una partición del conjunto de nodos en S y $\overline{S} = (N - S)$. Esta partición a su vez determina un conjunto de arcos, de los cuales unos tienen la propiedad de tener el inicio en S y el fin en $\overline{S}$ estos arcos se encuentran en $(S, \overline{S})$, mientras que el resto de estos arcos tienen su inicio en $\overline{S}$ y el fin en S estos arcos se encuentran en $(\overline{S}, S)$.

A continuación definimos un tipo especial de corte que depende de dos nodos s y t llamado **corte** s–t.

Definición 2.3.1 (corte s–t)
Dados una Red $G = (N, A)$; dos nodos s y t, un **corte** s–t *es cualquier* **corte** *donde $s \in S$ y $t \in \overline{S}$ ($s \notin \overline{S}$ y $t \notin S$)*

Observaciones:

- A un **corte** s–t también se le conoce como **corte que separa** s **de** t.
- Dado un par de nodos s y t es claro que existe un número finito de **cortes** s–t

De la definición anterior, un **corte** s–t está caracterizado por la partición del conjunto de nodos N en los subconjuntos S y $\overline{S} = (N - S)$, donde $s \in S$ y $t \in \overline{S}$.

Notación: Un corte que separa s y t será denotada por $[S, \overline{S}]$, donde evidentemente $s \in S$ y $t \in \overline{S}$.

Entonces cada vez que nos referimos a un **corte** s–t lo haremos mediante: $[S, \overline{S}]$, cada **corte** s–t $[S, \overline{S}]$, determina de modo único los siguientes conjuntos de arcos: (los cuales ya fueron definidos en la ecuación (2.2))

Arcos que salen de S y entran a $\overline{S}$ ó arcos **hacia adelante:**

$$(S,\overline{S}) := \left\{(i,j) \in A \Big/ \quad i \in S\ ,\ j \in \overline{S}\right\} \tag{2.11}$$

Arcos que salen de $\overline{S}$ y entran a S ó arcos **hacia atras:**

$$(\overline{S},S) := \left\{(i,j) \in A \Big/ \quad i \in \overline{S}\ ,\ j \in S\right\}. \tag{2.12}$$

Veamos a continuación el siguiente ejemplo para ilustrar las definiciones anteriores.

Ejemplo 2.8 *Consideremos la red de la figura (2.1), donde* $s = 1$ *y* $t = 6$, *entonces* $S = \{1,2,3\}$ *y* $\overline{S} = \{4,5,6\}$ *constituye un* **corte** *s–t (corte que separa s y t). Similarmente* $T = \{1,3,5\}$ *y* $\overline{T} = \{2,4,6\}$, *también es un* **corte** *s–t; tal como se muestra en la figura (2.4)*

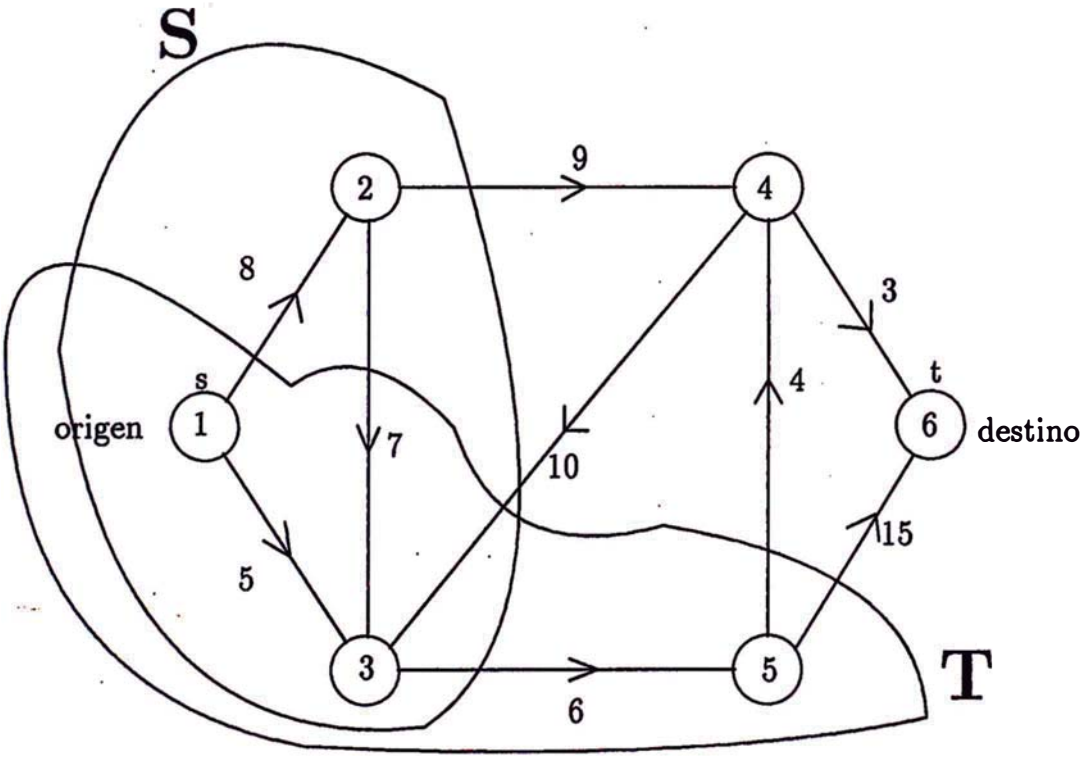


Figura 2.4: Cortes $[S,\overline{S}]$ y $[T,\overline{T}]$ que separan $s = 1$ y $t = 6$

En la red anterior, para el corte $[S, \overline{S}]$ se tiene el conjunto de arcos **hacia adelante** $(S, \overline{S}) = \{(2,4),(3,5)\}$ y el conjunto de arcos **hacia atras** $(\overline{S}, S) = \{(4,3)\}$. Mientras que para el corte $[T, \overline{T}]$ se tiene el conjunto de arcos **hacia adelante** $(T, \overline{T}) = \{(1,2),(5,4),(5,6)\}$ y el conjunto de arcos **hacia atras** $(\overline{T}, T) = \{(2,3),(4,3)\}$

Definición 2.3.2 (Capacidad de un corte s–t)
Dado el **corte** s–t $[S, \overline{S}]$, *definimos la capacidad* $u[S, \overline{S}]$ *de dicho corte como la suma de las capacidades de los arcos* **hacia adelante**, *es decir*

$$u[S, \overline{S}] := \sum_{(i,j) \in (S, \overline{S})} u_{ij} \tag{2.13}$$

Sea x es un flujo que respeta u, en una red capacitada; como $x_{ij} \leq u_{ij} \quad \forall\, (i,j) \in A$ entonces $\sum_{(i,j) \in (S, \overline{S})} x_{ij} \leq \sum_{(i,j) \in (S, \overline{S})} u_{ij} = u[S, \overline{S}]$, la expresión del lado izquierdo es la máxima cantidad de flujo que puede ser enviado de los nodos en S hacia los nodos en $\overline{S}$. Según esto la capacidad de un corte s–t es una cota superior para la cantidad máxima de flujo que puede ser enviado de S hacia $\overline{S}$.

Definición 2.3.3 (Corte Mínimo)
Se llama así a aquel corte s–t cuya capacidad es mínima entre todos los cortes s–t

Red Residual

Este concepto será muy útil para el desarrollo y descripción de los algoritmos para resolver el problema del flujo máximo; a continuación damos la definición formal y su interpretación en término de redes.

Dados una red $G = (N, A)$, una función capacidad $u : A \longrightarrow \mathbb{Z}^+$ y un flujo $x : A \longrightarrow \mathbb{Z}^+$; definimos la **capacidad residual** como una función r sobre el conjunto de arcos definido mediante:

$$\begin{aligned} r: \quad A &\longrightarrow \mathbb{Z}^+ \\ (i,j) &\longrightarrow r(i,j) := r_{ij} = (u_{ij} - x_{ij}) + x_{ji} \end{aligned} \tag{2.14}$$

Por definición $r_{ij} \geq 0$; en términos prácticos r_{ij} representa el flujo adicional máximo que aún puede pasar a través de los arcos (i,j) y (j,i)

Definición 2.3.4 (Red residual)

Dados una red $G = (N, A)$ con capacidad u, un flujo x de s a t, definimos y denotamos la red residual mediante:

$$G(x) = (N, \widehat{A}) \quad donde \quad \widehat{A} = \{(i,j) \in A / \; r_{ij} > 0\}$$

Es decir la red residual ($G(x)$) es una red que tiene los mismos nodos que G y arcos cuya capacidad residual es estrictamente positiva ($r_{ij} > 0$)

Observaciones:

a) Si $x = 0$ entonces $r_{ij} = u_{ij}$ $\forall$ $(i,j) \in A$, en consecuencia la red residual para este flujo coincide con la red original, es decir, $G(0) = G$

b) $$\begin{cases} r_{ij} &= u_{ij} - x_{ij} + x_{ji} \\ r_{ji} &= u_{ji} - x_{ji} + x_{ij} \end{cases} \implies r_{ij} + r_{ji} = u_{ij} + u_{ji} = \text{constante}$$

c) Dados una red capacitada $G = (N, A)$ y un flujo x, existe una correspondencia biunívoca entre la red original (G) y la red residual ($G(x)$) en el siguiente sentido:

- Dados $G = (N, A)$, u y x; $G(x)$ es obtenido mediante:

$$r_{ij} := (u_{ij} - x_{ij}) + x_{ji} \quad \text{mientras que}$$

- Dados $G(x)$ y u; x (y con él G) es obtenido mediante:

$$x_{ij} := \max_{(i,j) \in \widehat{A}} \{0, u_{ij} - r_{ij}\}$$

Definición 2.3.5 (Capacidad residual de un corte s–t)

Definimos la capacidad residual $r[S, \overline{S}]$ de un corte s–t como la suma de las capacidades residuales sobre los arcos hacia adelante del corte correspondiente es decir:

$$r[S, \overline{S}] := \sum_{(i,j) \in (S, \overline{S})} r_{ij} \tag{2.15}$$

Ejemplo 2.9 *Sean $G = (N, A)$ una red con capacidad u; y x un flujo; mostrados en la figura (2.5); se observa que existe una correspondencia entre G y $G(x)$*

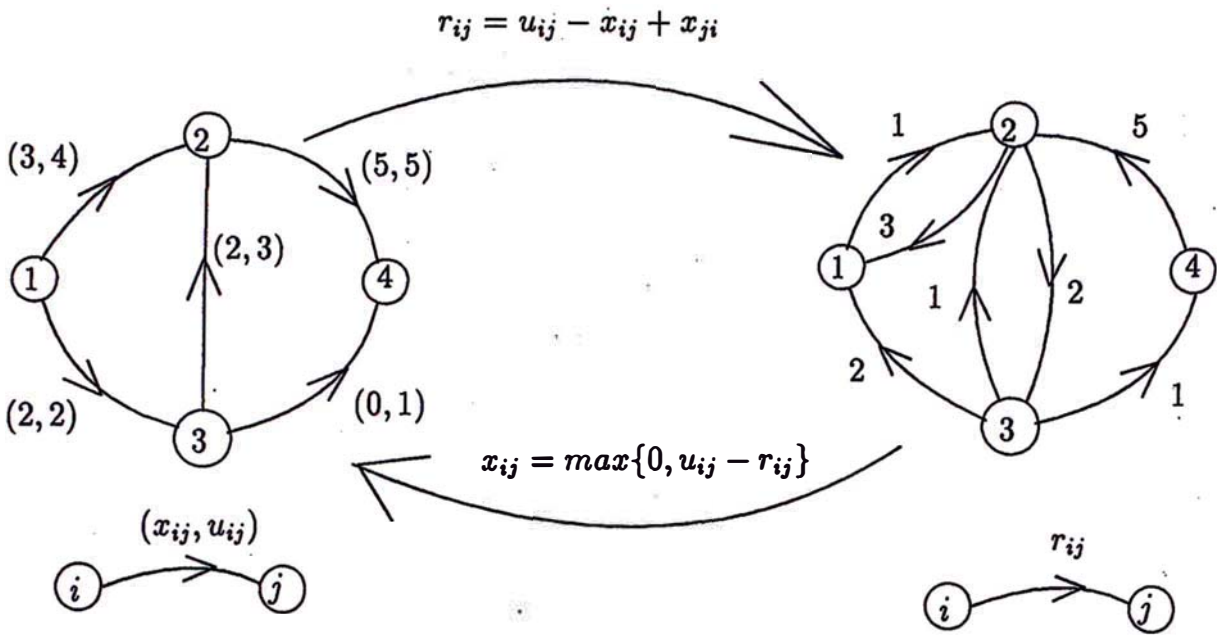


Figura 2.5: Correspondencia biunívoca entre las redes G y $G(x)$

La siguiente proposición relaciona los conceptos de corte $(S, \overline{S})$ y flujo x.

Proposición 2.1 *Sean $G = (N, A)$ una red capacitada; los nodos s (origen) y t (destino); un flujo x de s a t cuyo valor es $v(x)$ y $[S, \overline{S}]$ un corte que separa s y t entonces:*

$$v(x) = \sum_{i \in S} \left[\sum_{\{j/\,(i,j) \in A\}} x_{ij} - \sum_{\{j/\,(j,i) \in A\}} x_{ji} \right] = \sum_{(i,j) \in (S,\overline{S})} x_{ij} - \sum_{(i,j) \in (\overline{S},S)} x_{ij}$$

Prueba: Considerando que $S = \{s\} \cup \{S \setminus \{s\}\}$ podemos escribir la siguiente expresión:

$$\sum_{i \in S} \left[\sum_{\{j/\,(i,j) \in A\}} x_{ij} - \sum_{\{j/\,(j,i) \in A\}} x_{ji} \right] =$$

$$= \sum_{i=s} \left[\sum_{\{j/\,(i,j) \in A\}} x_{ij} - \sum_{\{j/\,(j,i) \in A\}} x_{ji} \right] + \sum_{i \in S \setminus \{s\}} \left[\sum_{\{j/\,(i,j) \in A\}} x_{ij} - \sum_{\{j/\,(j,i) \in A\}} x_{ji} \right]$$

$$= \underbrace{\left(\sum_{\{j/\,(s,j)\in A\}} x_{sj} - \sum_{\{j/\,(j,s)\in A\}} x_{js} \right)}_{v(x)} + \underbrace{\sum_{i\in S\setminus\{s\}} \left[\sum_{\{j/\,(i,j)\in A\}} x_{ij} - \sum_{\{j/\,(j,i)\in A\}} x_{ji} \right]}_{0}$$

(definición de valor del flujo x) (condición de equilibrio de masa para cada nodo $i \in S \setminus \{s\}$)

$$= v(x) + 0$$

$$= v(x)$$

Teorema 2.2 *El valor de cualquier flujo es menor o igual a la capacidad de cualquier corte en la red*

Prueba: Sean x cualquier flujo de s a t cuyo valor es $v(x)$ y $[S, \overline{S}]$ cualquier corte $s-t$; en el resultado de la proposición anterior (2.1) considerando $x_{ij} \leq u_{ij}$ en la primera sumatoria y $x_{ij} \geq 0$ enla segunda sumatoria, obtenemos:

$$\begin{aligned} v(x) = \sum_{(i,j)\in(S,\overline{S})} x_{ij} - \sum_{(i,j)\in(\overline{S},S)} x_{ij} &\leq \sum_{(i,j)\in(S,\overline{S})} u_{ij} - 0 \\ &= \sum_{(i,j)\in(S,\overline{S})} u_{ij} \\ &= u[S, \overline{S}] \end{aligned}$$

Luego se tiene el resultado: para todo flujo x y para todo corte $[S, \overline{S}]$

$$v(x) \leq u[S, \overline{S}] \tag{2.16}$$

Corolario 2.3 *si x_0 es un flujo de s a t y $[S_0, \overline{S_0}]$ es un corte s-t donde*

$$v(x_0) = u[S_0, \overline{S_0}]$$

entonces x_0 es un flujo cuyo valor es máximo y $[S_0, \overline{S_0}]$ es un corte cuya capacidad es mínima.

Prueba: Por el teorema anterior, para cualquier flujo x y cualquier corte $[S, \overline{S}]$ se tiene:

$$v(x) \leq u[S_0, \overline{S_0}] = v(x_0) \leq u[S, \overline{S}]$$

Luego:

$$v(x) \leq v(x_0) \quad \forall \text{ flujo } x \implies x_0 \text{ es un flujo cuyo valor es máximo} \quad \text{y}$$

$$u[S_0, \overline{S_0}] \leq u[S, \overline{S}] \quad \forall \text{ corte } [S, \overline{S}] \implies u[S_0, \overline{S_0}] \text{ es un corte cuya capacidad es mínima}$$

Corolario 2.4 *Si $[S, \overline{S}]$ es un corte que separa s de t, donde se cumple que*

$$r_{ij} = 0 \qquad \forall (i,j) \in (S, \overline{S})$$

entonces $[S, \overline{S}]$ es mínimo, además x calculado según $x_{ij} := max\{u_{ij} - r_{ij}, 0\}$ es un flujo de s a t cuyo valor es máximo

Prueba Sea (i,j) cualquier arco de $(S, \overline{S})$, entonces cumplen las siguientes relaciones:

$$r_{ij} = (u_{ij} - x_{ij}) + x_{ji} \tag{2.17}$$

$$x_{ij} \leq u_{ij} \qquad \text{y} \tag{2.18}$$

$$x_{ji} \geq 0 \tag{2.19}$$

entonces la condición $r_{ij} = 0$ conjuntamente con las relaciones (2.17), (2.18) y (2.19) implican que: $(u_{ij} - x_{ij}) = 0$ y $x_{ji} = 0$ para todo arco (i,j) de $(S, \overline{S})$, es decir

$$x_{ij} = u_{ij} \qquad \text{para todo arco } (i,j) \text{ de } (S, \overline{S}) \qquad \text{y} \tag{2.20}$$

$$x_{ij} = 0 \qquad \text{para todo arco } (i,j) \text{ de } (\overline{S}, S) \tag{2.21}$$

Sabemos que la capacidad del corte $[S, \overline{S}]$ por definición está dado por:

$$u[S, \overline{S}] = \sum_{(i,j) \in (S, \overline{S})} u_{ij}$$

con la ayuda de la ecuaciones (2.20) y (2.21) podemos escribir:

$$\begin{aligned} u[S, \overline{S}] &= \sum_{(i,j) \in (S, \overline{S})} u_{ij} \\ &= \sum_{(i,j) \in (S, \overline{S})} x_{ij} \\ &= \sum_{(i,j) \in (S, \overline{S})} x_{ij} - \underbrace{0} \\ &= \sum_{(i,j) \in (S, \overline{S})} x_{ij} - \sum_{(i,j) \in (\overline{S}, S)} x_{ij} \end{aligned}$$

Luego entonces:

$$u[S,\overline{S}] = \sum_{(i,j)\in(S,\overline{S})} x_{ij} - \sum_{(i,j)\in(\overline{S},S)} x_{ij} \tag{2.22}$$

La proposición (2.1) establece que:

$$v(x) = \sum_{(i,j)\in(S,\overline{S})} x_{ij} - \sum_{(i,j)\in(\overline{S},S)} x_{ij}$$

Entonces reemplazando esta expresión en la ecuación (2.22) y obtenemos:

$$v(x) = u[S,\overline{S}]$$

Por lo tanto hemos encontrado un flujo x y un corte $[S,\overline{S}]$, donde el valor del flujo es igual a la capacidad del corte, por el corolario (2.3) x es un flujo cuyo valor es máximo y $[S,\overline{S}]$ es un corte cuyo valor es mínimo, lo cual demuestra el corolario.

Definición 2.3.6 *Dados dos flujos x e y cuyos valores son respectivamente $v(x)$ y $v(y)$ decimos que y* **mejora** *x si $v(y) > v(x)$, dicha mejora es denotada por Δv con $\Delta v = v(y) - v(x) > 0$*

Los algoritmos que estudiaremos a continuación trabajan del siguiente modo: en cada iteración se tiene un flujo x de valor $v(x)$ y en el siguiente paso se trata de mejorar este flujo, es decir construir otro flujo y con valor $v(y)$ tal que $v(y) > v(x)$, el siguiente corolario establece una cota superior para Δv.

Corolario 2.5 *Si x es un flujo de valor $v(x)$, entonces el flujo adicional que aún puede ser enviado del origen s al destino t está limitado superiormente por la capacidad residual de cualquier corte*

Prueba: Si x es un flujo cuyo valor es $v(x)$; sea y un flujo que mejora x es decir: $v(y) = v(x) + \Delta v$ para algún $\Delta v > 0$.
Por el teorema (2.2) para el flujo y y para todo corte $[S,\overline{S}]$ se tiene:

$$v(y) \leq u[S,\overline{S}]$$

es decir,

$$v(x) + \Delta v \leq \sum_{(i,j)\in(S,\overline{S})} u_{ij} \tag{2.23}$$

y considerando $v(x) = \sum_{(i,j)\in(S,\overline{S})} x_{ij} - \sum_{(i,j)\in(\overline{S},S)} x_{ij}$, obtenemos:

$$\Delta v \leq \sum_{(i,j)\in(S,\overline{S})} (u_{ij} - x_{ij}) + \sum_{(i,j)\in(\overline{S},S)} x_{ij} \tag{2.24}$$

Además $\sum_{(i,j)\in(\overline{S},S)} x_{ij} = \sum_{(i,j)\in(S,\overline{S})} x_{ji}$ y reemplazando en la ecuación anterior (2.24) obtenemos:

$$\begin{aligned} \Delta v &\leq \sum_{(i,j)\in(S,\overline{S})} (u_{ij} - x_{ij}) + \sum_{(i,j)\in(S,\overline{S})} x_{ji} \\ &= \sum_{(i,j)\in(S,\overline{S})} (u_{ij} - x_{ij} + x_{ji}) \\ &= r[S,\overline{S}] \end{aligned}$$

Por lo tanto, $\Delta v \leq r[S,\overline{S}]$ es decir que la cantidad de flujo adicional Δv no puede exceder de $r[S,\overline{S}]$ para todo corte $[S,\overline{S}]$.

2.4 Método de Caminos Aumentantes

A seguir describimos el método mas sencillo e intuitivo para resolver el problema del flujo máximo, este método esta aún en un lenguaje de muy alto nivel (en el sentido que no se dan los detalles de implementación), estos detalles serán dados en la subseccion (2.5).

Como mencionamos antes el método maneja en cada paso un flujo empezando con el flujo cero, el cual debe ser mejorado en la siguiente iteración, dicha mejora se hace mediante un **camino aumentante** el cual se define a continuación

Definición 2.4.1 (Camino aumentante)
Dados una red $G = (N, A)$, dos nodos especiales s y t y un flujo x de s a t, definimos un **camino aumentante** *como un camino de s a t en la red residual $G(x)$.*

Definición 2.4.2 (Capacidad de un camino aumentante)
Dado un camino aumentante P, definimos su capacidad $u(P)$ como:

$$u(P) = \min\{r(i,j) \,/\, (i,j) \in AG(x)\,,\ \ (i,j) \in AP\}$$

Donde $AG(x)$ y AP denotan respectivamente las aristas de $G(x)$ y del camino P. La capacidad de un camino aumentante P (la cual por definición (2.4.1) está en la red residual) indica la cantidad de flujo adicional que aún puede ser enviado en la red original para mejorar el flujo.

Definición 2.4.3 (Arco saturado)
Se dice que el arco (i,j) *está* **saturado** *si* $x_{ij} = u_{ij}$

Método Genérico de Caminos Aumentantes

Entrada: Una red capacitada $G = (N, A)$ con capacidad u y dos nodos distintos s y t (origen y destino respectivamente)

Salida: Un flujo x de s a t que respeta u y cuyo valor es máximo.

```
x := 0;
mientras existe un camino aumentante de s a t en G(x) hacer:
    Identificar un camino aumentante P en G(x)
    Calcular δ = min{r_ij / (i,j) ∈ P}
    para cada (i,j) ∈ P hacer:
        r_ij := r_ij − δ;
        r_ji := r_ji + δ
para cada arco (i,j) en A hacer:
    x_ij := max{u_ij − r_ij, 0}
Retornar x
```

Figura 2.6: Algoritmo genérico de caminos aumentantes

El método trabaja con la red residual esto es, en cada paso se trabaja directamente con r_{ij}, pero también se tiene implícitamente el flujo correspondiente el cual puede ser calculado mediante: $x_{ij} := max\{0, u_{ij} - r_{ij}\}$ en cada paso. Ilustramos el método anterior con el problema de flujo máximo planteado en la figura 2.7(a) con origen en el nodo 1 y destino en el nodo 4.

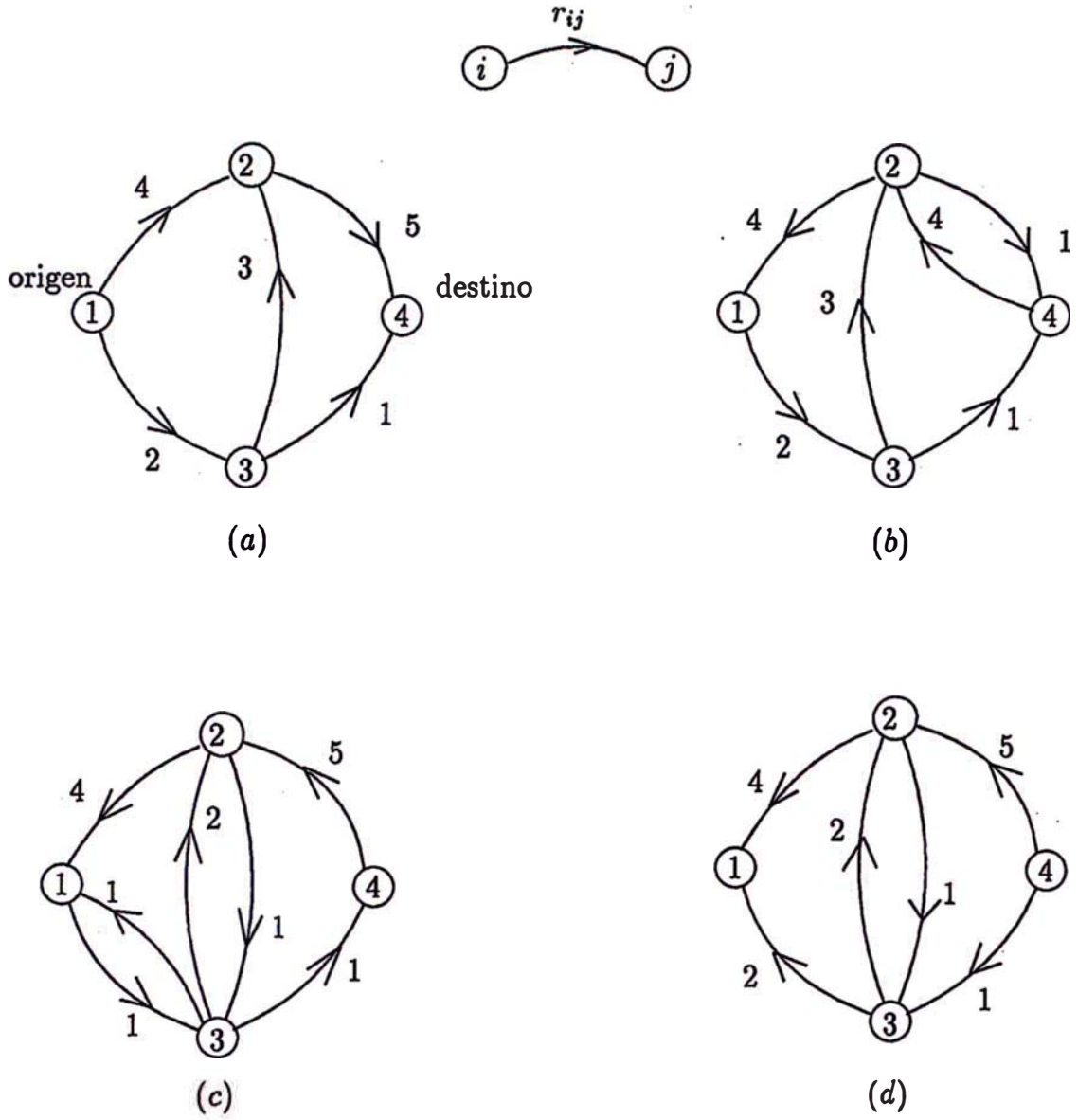


Figura 2.7: Ilustración del método genérico de caminos aumentantes (a) red residual para el flujo $x = 0$; (b) red residual luego de aumentar 4 unidades de flujo a través del camino 1—2—4; (c) red residual despues de enviar 1 unidad de flujo a través del camino 1—3—2—4 (d) red residual despues de enviar 1 unidad de flujo a través del camino 1—3—4, note que ya no existe camino de 1 hacia 4 en la red

Primera Iteración: El método inicializa el flujo a cero $x = 0$ y por lo tanto considera

la red residual $G(0) = G$, la cual se muestra en la figura 2.7(a). Elegimos el camino aumentante 1—2—4 cuya capacidad residual es $\delta = min\{r_{12}, r_{24}\} = min\{4, 5\} = 4$ y enviamos $\delta = 4$ unidades de flujo a través de este camino, con lo cual la capacidad residual del arco (1,2) se reduce a cero (por lo que es eliminado de la red residual) y la capacidad residual del arco reverso (2,1) es incrementado en 4 unidades (es decir $r_{12} := r_{12} - \delta = 4 - 4 = 0$ y $r_{21} := r_{21} + \delta = 0 + 4 = 4$). Por otra parte la capacidad residual del arco (2,4) queda reducido también en 4 unidades por lo que la nueva capacidad residual de este arco es 1 y la capacidad residual del arco reverso (4,2) es incrementado en 4 unidades, con lo cual la nueva capacidad residual del arco (4,2) es 4 (es decir $r_{24} := r_{24} - \delta = 5 - 4 = 1$ y $r_{42} := r_{42} + \delta = 0 + 4 = 4$). La red residual asi obtenida se muestra en la figura 2.7(b).

Segunda Iteración Elegimos ahora el camino aumentante 1—3—2—4 cuya capacidad residual es $\delta = min\{r_{13}, r_{32}, r_{24}\} = min\{2, 3, 1\} = 1$, por lo tanto enviamos $\delta = 1$ unidad de flujo a través de este camino, con lo cual la capacidad residual del arco (1,3) se reduce de 2 a 1 y la capacidad residual del arco (3,1) se incrementa de 0 a 1 (es decir, $r_{13} := r_{13} - \delta = 2 - 1 = 1$ y $r_{31} := r_{31} + \delta = 0 + 1 = 1$), realizando las mismas operaciones sobre los arcos (3,2) y (2,3) asi como sobre los arcos (2,4) y (4,2) obtenemos la red residual que se muestra en la figura 2.7(c)

Tercera Iteración Ahora se elige el camino aumentante 1—3—4 de capacidad residual $\delta = 1$ entonces enviamos 1 unidad de flujo a través de este camino. La capacidad residual del arco (1,3) se hace cero y la del arco (3,1) aumenta a 2; similarmente la capacidad residual del arco (3,4) se reduce a cero y la del arco (4,3) aumenta de 0 a 1, obteniendose la red residual de la figura 2.7(d)

Dado que en la red residual $G(x)$ (figura 2.7(d)) no existe camino de s a t entonces el método termina calculando el flujo para cada arco según la formula

$$x_{ij} := max\{0, u_{ij} - r_{ij}\}$$

de la siguiente manera:

Para el arco **(1,3)** tenemos $r_{13} = 0$ y $u_{13} = 4$ por lo tanto $x_{13} = 4$ y para el arco reverso

(3,1) tenemos $r_{31} = 4$ y $u_{31} = 0$ por lo tanto $x_{31} = 0$.
Por otra parte para el arco **(3,2)** tenemos $r_{32} = 1$ y $u_{32} = 0$ por lo tanto $x_{32} = 0$ y para el arco reverso **(2,3)** tenemos $r_{23} = 2$ y $u_{23} = 3$ por lo tanto $x_{23} = 1$.
Realizando los demás cálculos obtenemos: $x_{13} = 2$, $x_{34} = 1$ y $x_{24} = 5$ el cual según el algoritmo representa el flujo máximo (la veracidad de esta afirmación será demostrada a continuación), este flujo es mostrado en la figura (2.8).

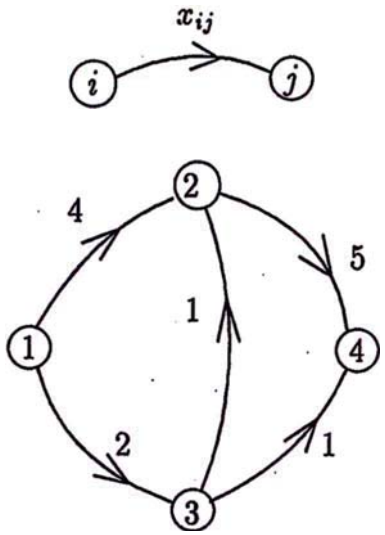


Figura 2.8: flujo máximo obtenido

2.5 Algoritmo de Etiquetas

En la sección anterior presentamos el método de caminos aumentantes sin preocuparnos que sea o no algoritmo, tampoco nos preocupamos por detalles importantes como por ejemplo ¿cómo establecer que en la red residual ya no existe un camino dirigido del origen al destino?, ¿cómo representar un camino aumentante en la red residual?, ¿cómo asegurar que el método termina en un número finito de iteraciones? (y por lo tanto el método se convierte en algoritmo) y cuando el algoritmo termina y devuelve un flujo ¿qué condición asegura que dicho flujo es el máximo? .

Todos estos y otros detalles serán útiles al momento de estudiar la correctitud y com-

plejidad de un tipo de implementación del método de caminos aumentantes conocido como el algoritmo de etiquetas que presentamos a continuación, este algoritmo no es polinomial como veremos al estudiar su complejidad, sin embargo en el siguiente capítulo estudiaremos tres implementaciones mas de este método los cuales si serán algoritmos polinomiales como veremos en su debida oportunidad.

2.5.1 Implementación

El algoritmo de etiquetas utiliza la técnica de búsqueda para identificar un camino dirigido en $G(x)$ del origen al destino (camino de aumento). El algoritmo irá etiquetando los vértices de acuerdo a cierto criterio, empezando desde el vértice origen, estableciendo así un camino dirigido en la red residual desde el origen hasta algún vértice que eventualmente podría ser el vértice destino en cuyo caso la búsqueda termina pues se ha logrado encontrar un camino aumentante.

En cualquier paso del algoritmo se tiene dos conjuntos bien definidos: el conjunto de vértices **etiquetados** y el conjunto de vértices **no etiquetados**. Los vértices etiquetados son aquellos que el proceso de búsqueda ha alcanzado. El algoritmo selecciona repetitivamente un vértice etiquetado y analiza los vértices de su lista de adyacencia para etiquetar otros vértices. Eventualmente el vértice destino será etiquetado y entonces el algoritmo envía el máximo flujo posible a través del camino del origen al destino determinado en el proceso de búsqueda. Luego elimina todas las etiquetas y el proceso se repite. El algoritmo termina cuando todos los vértices etiquetados han sido procesados y el vértice **destino** permanece sin etiqueta, es decir el vértice **destino** no puede ser etiquetado.

El algoritmo utiliza la función **pred** definido en el capítulo 1 para describir los caminos dirigidos en la red residual, asi como los conjuntos de nodos S (nodos etiquetados) y $LIST$ (nodos etiquetados que aún no fueron procesados completamente) A continuación presentamos el algoritmo:

Entrada: Una red $G = (N, A)$, una función capacidad $u :\longrightarrow \mathbb{Z}^+$ y dos nodos distintos s y t llamados origen y destino respectivamente.

Salida: Un flujo x de s a t que respeta u y cuyo valor es máximo.

```
r := u;
repetir
    pred(j) = 0 para cada j ∈ N
    S := {s}; LIST := {s};
    mientras (LIST ≠ φ) y (t ∉ S) hacer:
        retirar un elemento i de LIST;
        para cada (i, j) ∈ A(i) hacer:
            si (r_ij > 0) y (j ∉ S)
                entonces
                    pred(j) := i;
                    S := S ∪ {j};
                    LIST := LIST ∪ {j}
    si t ∈ S entonces
        use pred para encontrar un camino orientado P de s a t;
        δ = min{r_ij / (i, j) es arco de P};
        para cada arco (i, j) de P hacer:
            r_ij := r_ij − δ
            r_ji := r_ji + δ
hasta que LIST = φ;
para cada arco (i, j) ∈ A hacer:
    x_ij := max{u_ij − r_ij, 0}
```

Figura 2.9: Algoritmo de Etiquetas

Observación: Cada iteración del algoritmo anterior empieza con una función $r : A \longrightarrow \mathbb{Z}^+$, una parte S de N, una parte $LIST$ de N y una función $pred : S \setminus \{s\} \longrightarrow S$.

La primera iteración comienza con $r := u$, $S = LIST = \{s\}$ y $pred \equiv 0$. En cada iteración ocurre uno y solo uno de los siguientes casos:

Caso 1: (Intento de hallar camino): $LIST \neq \phi$ y $t \notin S$

En este caso seleccionamos un nodo etiquetado pero que aún no ha sido completamente procesado, es decir tomamos $i \in LIST$; y para cada nodo j adyacente a i (es decir para cada $(i, j) \in A(i)$) realizamos lo siguiente:

Si $r_{ij} > 0$ y $j \notin S$ (j aún no esta completamente procesado) entonces j hace parte del camino aumentante ($pred(j) = i$) y entonces j se acrescenta a S y a $LIST$.

Caso 2: (Aumento de flujo) $t \in S$

Usando *pred* determinamos un **camino aumentante** P y luego calculamos su **capacidad** ($\delta := min\{r_{ij} \ / \ (i,j)$ es arco de $P\}$), a continuación actualizamos las capacidades residuales de los arcos que se encuentran a lo largo de P, es decir $r_{ij} := r_{ij} - \delta$ y $r_{ji} := r_{ji} + \delta$.

Finalmente inicializamos S, $LIST$ y *pred* y comenzamos una nueva iteración

Caso 3: (No se puede etiquetar el destino) $LIST = \phi$ y $t \notin S$

Esto quiere decir que el flujo ya no puede ser mejorado, por lo tanto el algoritmo termina calculando el flujo según:

$x_{ij} := max\{u_{ij} - r_{ij}, 0\}$ para cada (i,j) arco de G

2.5.2 Correctitud y complejidad

A continuación estableceremos las razones por las cuales el flujo que se obtiene al terminar el algoritmo de etiquetas es máximo (correctitud) y cuánto tiempo tarda el algoritmo en resolver el problema (complejidad).

Correctitud

Para demostrar la correctitud del algoritmo, notemos que el algoritmo mantiene las siguientes propiedades invariantes al inicio de cada iteración,

a) un flujo x, definido mediante $x_{ij} := max\{u_{ij} - r_{ij}, 0\}$

b) para cada k en S, *pred* determina un camino orientado P de s a k, cada uno de cuyos arcos tiene capacidad residual positiva.

c) o el valor de x aumenta o el valor de x no se altera y el conjunto $S \setminus LIST$ aumenta.

La invariante *c*) establece que al inicio de cualquier iteración o el valor de x aumenta (caso que ocurre si se logra encontar un camino aumentante) o el valor de x no se altera, pero el conjunto $S \setminus LIST$ aumenta, en este caso aún no se ha logrado hallar un camino aumentante, pero el conjunto de nodos etiquetados completamente procesados S aumenta y el conjunto de nodos etiquetados aún no procesados $LIST$ disminuye entonces en algún

momento $LIST$ llega a ser vacío (con lo cual el algoritmo terminará en algún momento). Al término del algoritmo obtenemos un conjunto $S^* \subset N$ ($S^* \neq N$).

Supongamos que el algoritmo termina (hecho asegurado por la observación anterior), entonces quiere decir (según el **caso 3:** de la subseccion anterior 2.5.1), que el nodo destino t no puede ser etiquetado. Sea además el conjunto S^* que consiste de los nodos etiquetados, además $\overline{S^*} = N \setminus S^*$ es el conjunto de nodos no etiquetados, luego entonces es claro que $s \in S^*$ y $t \in \overline{S^*}$; es decir $[S^*, \overline{S^*}]$ es un corte que separa s y t. Consideremos también el flujo x^* obtenido al terminar el algoritmo, calculado según $x^*_{ij} := max\{u_{ij} - r_{ij}, 0\}$ para todo arco $(i,j) \in A$

Dado que t no puede ser etiquetado entonces debe ocurrir que:

$$r_{ij} = 0 \quad \text{para todo arco} \quad (i,j) \quad \text{de} \quad (S^*, \overline{S^*}) \tag{2.25}$$

pues de lo contrario, si existiese un arco $(i_0, j_0) \in (S^*, \overline{S^*})$ tal que $r_{i_0 j_0} > 0$ entonces el algoritmo aún no terminaría lo cual no puede ser, pues estamos asumiendo que el algoritmo terminó.

El corolario (2.4) y la ecuación (2.25) establecen que x^* es un flujo de s a t cuyo valor es máximo, lo cual muestra la correctitud del algoritmo de etiquetas.

Nótese que el algoritmo además de hallar el flujo máximo, nos devuelve un corte $[S^*, \overline{S^*}]$ cuya capacidad es mínima.

La prueba de correctitud del algoritmo anterior nos permite establecer dos resultados teóricos que enunciamos a continuación:

Teorema 2.6 (Teorema del Flujo Máximo – Corte Mínimo) *Sean dados una red capacitada $G = (N, A)$, dos nodos: origen (s) y destino (t), entonces el valor del flujo máximo de s a t en es igual a la capacidad del corte mínimo*

Prueba:

Apliquemos el algoritmo de etiquetas descrito anteriormente a la red capacitada G con origen y destino s y t respectivamente. Entonces el algoritmo halla el flujo máximo x^*, pero también el algoritmo nos devuelve un conjunto de nodos S^*, el cual determina el corte $[S^*, \overline{S^*}]$.

La prueba de correctitud del algoritmo de etiquetas nos permite afirmar que $[S^*, \overline{S^*}]$ es el corte mínimo y mas aún también afirmamos que:

$$v(x^*) = u[S^*, \overline{S^*}]$$

Es decir, el valor del flujo máximo es igual a la capacidad del corte mínimo.

Observación: La prueba del teorema del flujo máximo–corte mínino, se basa fundamentalmente en la prueba de correctitud del algoritmo de etiquetas

Teorema 2.7 *Dados una red capacitada $G = (N, A)$, dos nodos: origen (s) y destino (t); Un flujo x^* en G es máximo si y solo si no existe ningun camino aumentante en la red residual $G(x^*)$*

Prueba:

Sea x^* un flujo de s a t en G

$\Longrightarrow$) Supongamos que $G(x^*)$ contiene un camino aumentante P, entonces x^* no puede ser el flujo máximo, pues $v(x^*)$ aún puede ser mejorado enviando cierta cantidad positiva de flujo a través del camino aumentante P.

$\Longleftarrow$) Si no existe camino aumentante en $G(x^*)$, quiere decir que el algoritmo de etiquetas terminó y nos devolvió el flujo x^* el cual es máximo; el hecho de que x^* sea máximo está basada también en la prueba de correctitud del algoritmo de etiquetas

Complejidad (en el peor caso)

Dado que el algoritmo básicamente encuentra caminos aumentantes y envia cierta cantidad de flujo a través de dichos caminos, la complejidad de tiempo $T(m, n)$ del algoritmo está dado por la siguiente expresión:

$$T(m, n) = (\text{complejidad de cada aumentación}) \times (\#\text{ de aumentaciones}) \qquad (2.26)$$

Por lo tanto es necesario calcular la complejidad de cualquier aumentación y estimar el número máximo de aumentaciones.

Para calcular la complejidad de una aumentación observemos que cualquier aumentación consta de dos operaciones: **determinación del camino aumentante** y envio

de δ unidades de flujo a través de dicho camino, entonces para calcular la complejidad de cualquier aumentación es necesario calcular las complejidades de las operaciones mencionadas.

- Si se utiliza la **búsqueda en largura** para determinar un camino dirigido (camino aumentante) en la red residual ésta operación tendrá complejidad $\mathcal{O}(m)$ en el peor caso.

- Por otra parte cualquier camino aumentante tendrá longitud m en el peor caso, por lo tanto la complejidad del envío de δ unidades de flujo a lo largo del camino aumentante hallado será $\mathcal{O}(m)$ en el peor caso.

En consecuencia la complejidad de cualquier aumentación está dada por: $\mathcal{O}(m)+\mathcal{O}(m) = \mathcal{O}(m)$. Por otra parte el número máximo de aumentaciones está acotado superiormente por nU (considerando que todos los arcos tienen capacidad U), es decir su complejidad es $\mathcal{O}(nU)$. Reemplazando en (2.26) obtenemos la complejidad de tiempo del algoritmo de caminos aumentantes, la cual está dada por:

$$T(m,n) = \mathcal{O}(m)\mathcal{O}(nU) = \mathcal{O}(mnU)$$

Esta complejidad no es polinomial, pues podría ocurrir por ejemplo que $U = 2^n$, fenómeno que estudiaremos en la siguiente sección.

2.5.3 Ventajas y desventajas

Entre las pocas ventajas que podemos mencionar para este algoritmo se encuentran su facilidad de implementación, su simplicidad y principalmente su utilidad en la demostración del teorema del flujo máximo–corte mínimo y el hecho de ser el punto de partida para el desarollo de otros algoritmos mas eficientes.

Por otra parte las desventajas mas saltantes son:

1. El algoritmo no es polinomial (su complejidad es $\mathcal{O}(mnU)$), esto ocurre principalmente cuando el valor de U es demasiado grande; por ejemplo si $U = 2^n$ el algoritmo sería exponencial y en consecuencia intratable en la práctica. Ilustramos este caso con el problema del flujo máximo propuesto en la figura (2.10).

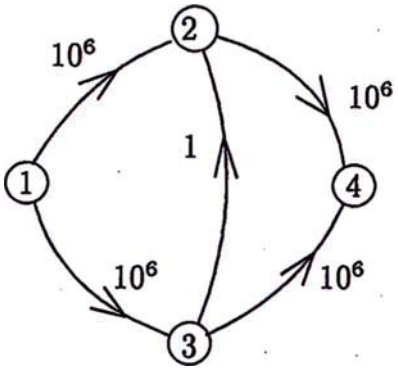
2. Si las capacidades son irracionales, el algoritmo podría no terminar. En algunos casos, el algoritmo no termina, pero los sucesivos valores de los flujos convergen a un valor estrictamente menor que el del flujo máximo. Este caso está fuera de nuestro análisis, pues en este trabajo estamos considerando capacidades en $\mathbb{Z}^+$.

3. El algoritmo es "olvidadizo" en el siguiente sentido: en cada iteración, el algoritmo genera una serie de etiquetas, los cuales contienen información acerca de los caminos aumentantes desde la fuente (origen) hacia los otros nodos. La implementación descrita elimina toda esta información al terminar una iteración, para iniciar la siguiente, realizando nuevamente los cálculos anteriores, cuando lo ideal sería utilizar la información entre una iteración y la siguiente.

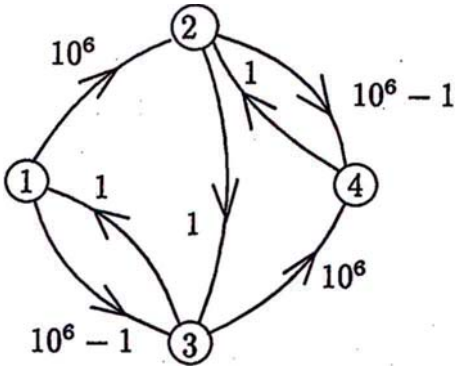
A continuación ilustramos por qué el algoritmo no es polinomial, consideremos el problema del flujo máximo dado en la figura (2.10 a) donde el nodo origen es 1 y el nodo destino es 4, nótese que ésta red tiene capacidades de orden bastante grandes (es decir U es grande);

Se ejecuta el algoritmo y en la primera iteración selecciona el camino aumentante 1-3-2-4, dado que la capacidad residual de dicho camino es $\delta = 1$, entonces el algoritmo envía 1 unidad de flujo a través de dicho camino; obteniéndose la red de la figura (2.10 b). En la segunda iteración el algoritmo selecciona el camino aumentante 1-2-3-4 y envía también 1 unidad de flujo a través de dicho camino obteniéndose la red de la figura (2.10 c). En la tercera iteración se elige el camino 1-3-2-4, en la cuarta iteración el camino 1-2-3-4 y asi sucesivamente; en general en las iteraciones impares elige el camino 1-3-2-4 y en las iteraciones pares elige el camino 1-2-3-4.

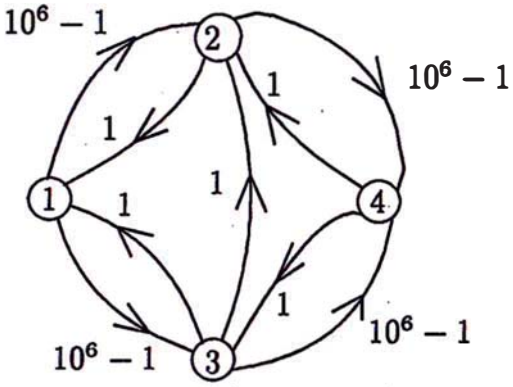
Podemos deducir para este caso especial que el número de iteraciones es igual a $2 \times 10^6 = 2 \times U = \mathcal{O}(10^6)$, luego el tiempo que toma al algoritmo resolver el problema será del orden $\mathcal{O}(10^6) = \mathcal{O}(U)$, lo cual es demasiado alto.



(a)



(b)



(c)

etc.

Figura 2.10: (a) problema del flujo máximo dado con origen en 1 y destino en 4, (b) red residual luego de enviar una unidad de flujo a través del camino aumentante 1-3-2-4; (c) red residual luego de enviar una unidad de flujo a través del camino aumentante 1-2-3-4; el algoritmo continua asi hasta saturar todas las capacidades

Esto se debe a que la complejidad del algoritmo ($\mathcal{O}(mnU)$), depende explícitamente del número U sobre el cual no tenemos ningun control, pues se trata de un dato proporcionado por el problema práctico que se está resolviendo. Entonces el objetivo será diseñar algoritmos cuya complejidad sea mas manejable y para eso la complejidad debe depender de $Log(U)$ en lugar de depender de U o mejor aún ni siquiera depender de U. Este es el motivo del siguiente capítulo donde desarrollaremos algoritmos polinomiales.

Capítulo 3

ALGORITMOS POLINOMIALES

En este capítulo presentamos los llamados algoritmos polinomiales que surgen a partir de la necesidad de mejorar el tiempo de ejecución en el peor caso ($\mathcal{O}(mnU)$) del algoritmo genérico visto en el capítulo anterior. Presentamos en particular los algoritmos "Capacity scaling", "Shortest augmenting path" y "Preflow push", siendo el último de los mencionados el más eficiente tanto teóricamente como en aplicaciones. Como siempre para cada uno de ellos presentamos la descripción del método, el algoritmo y asimismo una prueba de correctitud y un análisis de complejidad, también se resaltan las ventajas y desventajas de cada uno de ellos.

3.1 Algoritmo "Capacity Scaling"

El algoritmo de etiquetas escoge arbitrariamente los caminos aumentantes, razón por lo cual el número de aumentaciones ($\mathcal{O}(nU)$) puede resultar en algunos casos extremadamente alto si el valor de U es grande, tal como se observó en el capítulo anterior.

Entonces la idea del algoritmo "Capacity Scaling" es reducir el número de aumentaciones, para lo cual escoge los caminos aumentantes con muchísimo mas cuidado (allí radica su mejora con respecto al algoritmo anterior).

La idea esencial de este algoritmo es escoger caminos aumentantes cuyas capacidades sean *suficientemente grandes*, pues como veremos a continuación este tipo de caminos puede ser encontrado eficientemente (es decir en tiempo razonablemente corto).

Definición 3.1.1 (Δ–red residual)

Sean dados una red capacitada $G = (N, A)$, dos nodos s y t (origen y destino resp.) y un flujo x de s a t; entonces definimos para cada $\Delta > 0$ la **Δ–red residual** *como el subgrafo de $G(x)$ denotado por $G(x, \Delta)$ y definido mediante:*

$$G(x, \Delta) = (N, \widehat{A}\,)$$

donde,

$$\widehat{A} = \{(i, j) \in A \,/\; r_{ij} \geq \Delta\}$$

Es decir, $G(x, \Delta)$ es un subgrafo de $G(x)$ con el mismo conjunto de nodos (N), pero cuyos arcos son aquellos cuya capacidad residual es mayor o igual a Δ.

Ilustramos ésta definición en la figura (3.1); donde en (a) se tiene una red residual $G(x)$ para algún flujo x; mientras que en (b) tenemos un subgrafo $G(x, \Delta)$ de $G(x)$, donde el conjunto de nodos es el mismo para ambas redes, pero los arcos en $G(x, \Delta)$ son aquellos arcos de $G(x)$ cuya capacidad residual es mayor o igual a $\Delta = 8$ en este caso particular.

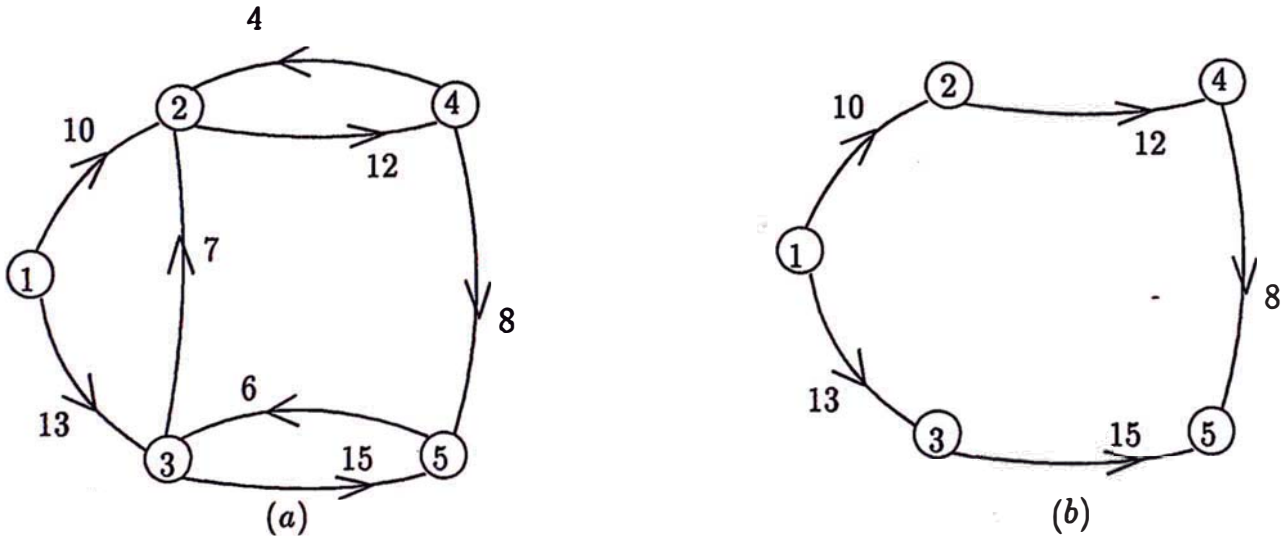


Figura 3.1: (a) red residual $G(x)$; (b) Δ–red residual $G(x, \Delta)$, con $\Delta = 8$

Observación: De la definición se tiene que: $G(x, 1) = G(x)$

3.1.1 Implementación

El algoritmo es muy semejante al algoritmo de etiquetas visto en el capítulo anterior; la diferencia está en que mientras el algoritmo de etiquetas escoge los caminos aumentantes arbitrariamente, el algoritmo "capacity scaling" escoge caminos aumentantes con capacidad residual suficientemente grandes.

Para ello considera un valor inicial para **delta** tal como Δ_0 y utilizando la Δ_0–red residual envía todo el flujo posible a través de los caminos aumentantes que existen en dicha Δ_0–red residual; si ya no existe ningún camino aumentante en la Δ_0–red residual, entonces se considera un nuevo valor para **delta** tal como Δ_1 consistente en el 50% del valor anterior (es decir $\Delta_1 =$ [illegible] y el proceso continúa con la nueva Δ_1–red residual. Todo este proceso a su vez continúa hasta que el valor de **delta** sea igual a 1, (es decir hasta que $\Delta = 1$).

A continuación presentamos el algoritmo:

Entrada: Una red $G = (N, A)$, una función capacidad $u :\longrightarrow \mathbb{Z}^+$; dos nodos distintos s y t llamados origen y destino respectivamente y el número U.

Salida: Un flujo x de s a t que respeta u y cuyo valor es máximo.

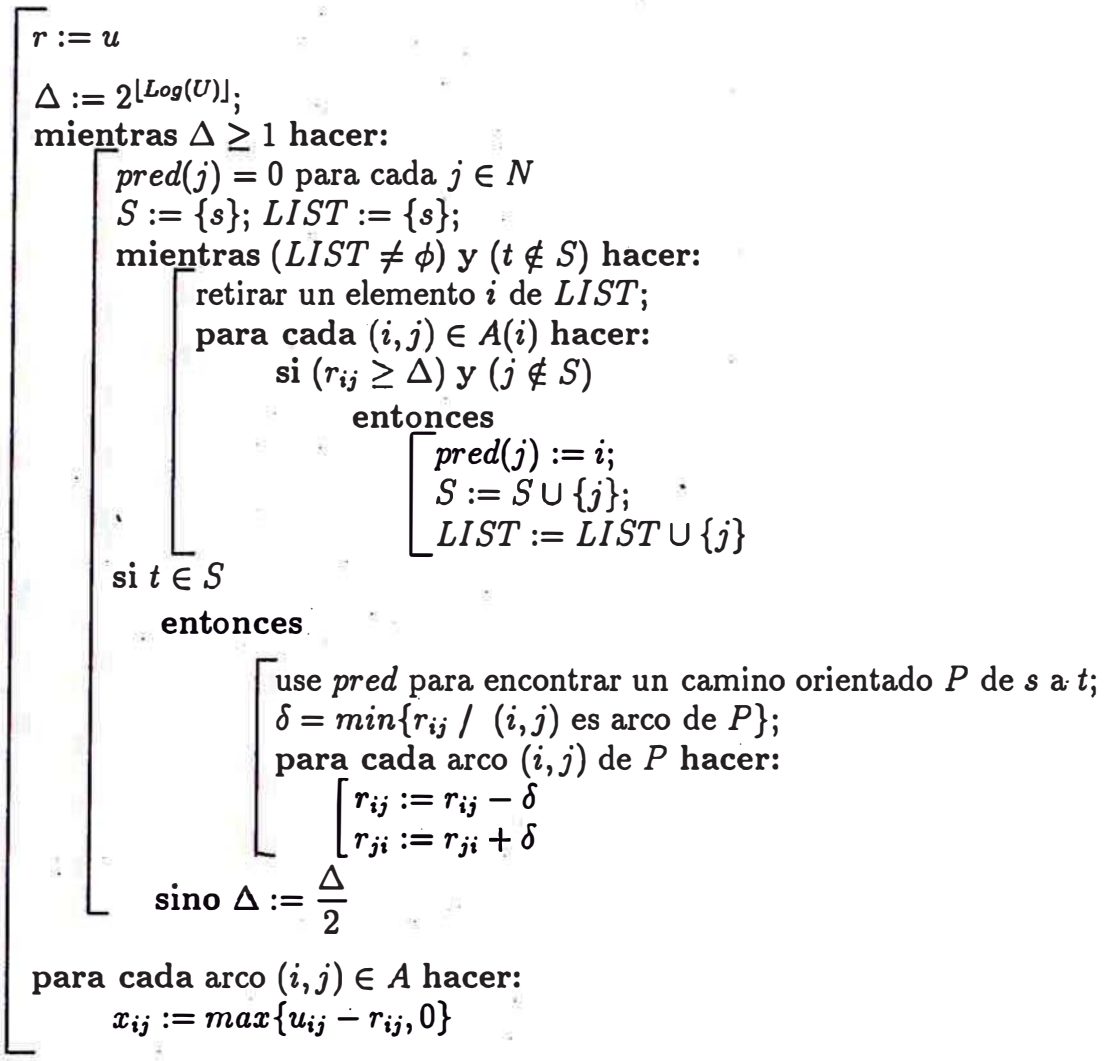


Figura 3.2: Algoritmo "Capacity Scaling"

Nota: El logaritmo que se menciona en el algoritmo es el logaritmo en base 2.

Observación: Cada iteración del algoritmo "Capacity Scaling" empieza con una función $r : A \longrightarrow \mathbb{Z}^+$, un entero Δ una parte S de N, una parte $LIST$ de N y una función $pred : S \setminus \{s\} \longrightarrow S$.

La primera iteración comiénza con $r := u$, $S = LIST = \{s\}$ y $pred \equiv 0$. En cada iteración ocurre uno y solo uno de los siguientes casos:

Caso 1: (Intento de hallar camino de aumento): $\Delta \geq 1$; $LIST \neq \phi$ y $t \notin S$

En este caso seleccionamos un nodo etiquetado pero que aún no ha sido completamente procesado, es decir tomamos $i \in LIST$; y para cada nodo j adyacente

a i (es decir para cada $(i,j) \in A(i)$) realizamos lo siguiente:

Si $r_{ij} \geq \Delta$ y $j \notin S$ (j aún no esta completamente procesado) entonces j hace parte del camino aumentante ($pred(j) = i$) y entonces j se acrescenta a S y a $LIST$.

Caso 2: (Aumento de flujo) $\Delta \geq 1$ y $t \in S$

Usando *pred* determinamos un **camino aumentante** P y luego calculamos su **capacidad** ($\delta := min\{r_{ij} \ / \ (i,j)$ es arco de $P\}$), a continuación actualizamos las capacidades residuales de los arcos que se encuentran a lo largo de P, es decir $r_{ij} := r_{ij} - \delta$ y $r_{ji} := r_{ji} + \delta$.

Finalmente inicializamos S, $LIST$ y *pred* y comenzamos una nueva iteración.

Caso 3: (Disminuye Δ) $\Delta \geq 1$; $LIST = \phi$ y $t \notin S$

Hacemos $\Delta = \frac{\Delta}{2}$; $S = LIST = \{s\}$ y $pred(s) = 0$ y empezamos una nueva iteración.

Caso 4: (No se puede etiquetar el destino ni disminuir *delta*) $\Delta < 1$

Esto quiere decir que el flujo ya no puede ser mejorado, por lo tanto el algoritmo termina calculando el flujo según:

$x_{ij} := max\{u_{ij} - r_{ij}, 0\}$ para cada (i,j) arco de G

Ejemplo 3.1 *A modo de ilustrar el algoritmo, apliquemos a la red de la figura 2.10 (a) donde el algoritmo de etiquetas realizó* 2×10^6 *pasos para hallar el flujo máximo*

solución: En este caso $U = 10^6$, entonces $\Delta = 2^{\lfloor Log(U) \rfloor} = 2^{\lfloor Log(10^6) \rfloor} = 2^{\lfloor 19.9316 \rfloor} = 2^{19}$. Entonces en la Δ-red residual solo consideramos arcos cuya capacidad residual sea mayor o igual a $\Delta = 2^{19}$, como $10^6 > 2^{19}$ entonces consideramos la red mostrada en la siguiente figura:

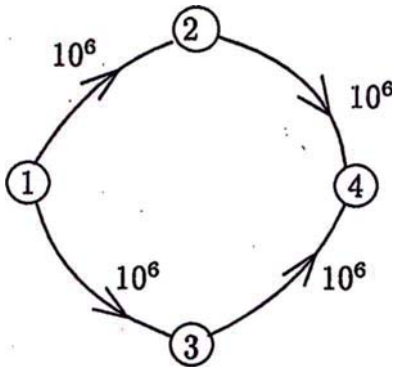


Figura 3.3: 2^{19}-red residual

elegimos los caminos 1-2-4 y 1-3-4 y enviamos $\delta = 10^6$ unidades de flujo a través de cada uno de dichos caminos, con lo cual ya se calculó el flujo máximo, pero el algoritmo hace $\Delta = \frac{\Delta}{2} = \frac{2^{19}}{2} = 2^{18}$ y considera la 2^{18}-red residual, dado que ya no existe ningun camino aumentante entonces el valor de delta se reduce en 50% cada vez hasta llegar al valor de 1, donde el algoritmo termina. Concluimos observando que el algoritmo "Capacity scaling" realizó tan solo 2 aumentaciones en tanto que el algoritmo anterior tuvo que realizar 2×10^6 aumentaciones. También debe notarse que el algoritmo "Capacity scaling" realizó además de las 2 aumentaciones, 19 iteraciones más para llegar al valor de $\Delta = 1$, con lo cual termina el algoritmo, sin embargo 19+2=21 iteraciones supera ampliamente a las 2×10^6 iteraciones del algoritmo de etiquetas.

3.1.2 Correctitud y Complejidad

Correctitud

La prueba de correctitud del algoritmo es inmediato, observando que el valor de Δ siempre llega hasta 1, cuyo caso sería exactamente el último paso del algoritmo de etiquetas visto en el capítulo anterior, en donde no existe camino aumentante la red $G(x,1) = G(x)$ y por lo tanto el flujo obtenido es máximo.

Complejidad

Llamamos **fase** al conjunto de pasos efectuados por el algoritmo, durante las cuales Δ permanece constante. Si se quiere explicitar el valor de Δ nos referiremos como Δ–**fase**.

Proposición 3.1 *Las siguientes afirmaciones son verdaderas:*

a) *El número de* **fases** *es* $\mathcal{O}(Log\ U)$

b) *El algoritmo "Capacity Scaling" efectúa a lo más* $2m$ *aumentaciones en cada* **fase**

Prueba:

a) El algoritmo empieza con $\Delta = 2^{\lfloor Log\ U \rfloor}$, luego disminuye este valor a la mitad en la siguiente iteración y luego nuevamente a la mitad en la siguiente iteración y asi sucesivamente hasta llegar al valor de $\Delta = 1$, utilizando el mismo argumento de la búsqueda binaria establecemos que el número de iteraciones o el número de **fases** es $1 + \lfloor Log\ U \rfloor = \mathcal{O}(Log\ U)$.

b) Consideremos dos **fases** consecutivos cualesquiera; sea la Δ–**fase** aquella fase que está culminando, con su respectivo flujo (sea tal flujo x' cuyo valor es v'); entonces la próxima fase (si existe) será la $\frac{\Delta}{2}$–**fase**.

Consideremos el conjunto de nodos S definido mediante:

$$S = \{i \in N \,/\, \text{ existe un camino de } s \text{ a } i \text{ en } G(x', \Delta)\} \tag{3.1}$$

Dado que $G(x', \Delta)$ no contiene ningún camino aumentante del origen (s) al destino (t), (pues si existiera algún camino la Δ–**fase** aún no terminaría) entonces $t \notin S$, con lo cual se determina el corte $[S, \overline{S}]$ que separa s de t.

Por definición de S tenemos necesariamente que:

$$r_{ij} < \Delta \quad \forall\, (i,j) \in (S, \overline{S}) \tag{3.2}$$

De (3.2) tenemos:

$$\sum_{(i,j)\in(S,\overline{S})} r_{ij} < \sum_{(i,j)\in(S,\overline{S})} \Delta$$

$$\Longrightarrow \quad r[S, \overline{S}] < \Delta \sum_{(i,j)\in(S,\overline{S})} 1 < \Delta(m)$$

Es decir,

$$r[S, \overline{S}] < m\Delta \tag{3.3}$$

Esto quiere decir que la capacidad residual del corte $[S, \overline{S}]$ es a lo más $m\Delta$.

Si x^* es el flujo máximo y v^* su valor, entonces por el corolario (2.5), el flujo adicional que aún puede ser enviado a partir de x' para llegar a x^* es menor o igual a la capacidad residual del corte $[S, \overline{S}]$ es decir, $\Delta v \leq r[S, \overline{S}]$ y conjuntamente con la ecuación (3.3) tenemos:

$$v^* - v' \leq m\Delta \tag{3.4}$$

Al terminar la Δ–**fase** empieza una nueva fase con $\Delta = \frac{\Delta}{2}$ ($\frac{\Delta}{2}$–**fase**), en ésta última fase cada aumentación puede llevar como mínimo $\frac{\Delta}{2}$ unidades de flujo.

En este punto nos hacemos la siguiente pregunta: ¿ cuál es el máximo número de aumentaciones que se pueden realizar en la $\frac{\Delta}{2}$–**fase**, para no violar la condición dada en (3.4)?

Como cada aumentación lleva como mínimo $\frac{\Delta}{2}$ unidades de flujo, entonces el número máximo de aumentaciones es $2m$, pues $\frac{\Delta}{2}(2m) = m\Delta$ que es el flujo adicional máximo que puede ser enviado (según 3.4).

Luego entonces, en cada **fase** se realizan como máximo $2m$ aumentaciones.

Teorema 3.2 *El algoritmo "Capacity Scaling" resuelve el problema del flujo máximo en tiempo* $\mathcal{O}(m^2 Log\ U)$

Prueba: La complejidad de tiempo del algoritmo está dado por:

$$T(m, n, U) = (\#\text{ de aumentaciones})(\text{ complejidad de cada aumentación}) \tag{3.5}$$

donde:

$$\text{complejidad de cada aumentación } = \mathcal{O}(m) \tag{3.6}$$

Además,

$$\#\text{ de aumentaciones } = (\#\text{ de fases }) \times (\#\text{ de aumentaciones en cada fase}) \tag{3.7}$$

De la proposición (3.1):

$$\begin{aligned} \#\text{ de fases} &= \mathcal{O}(Log\ U) \\ \#\text{ de aumentaciones en cada fase} &= 2m = \mathcal{O}(m) \end{aligned}$$

Reemplazando en (3.7):

$$\#\text{ de aumentaciones} = \mathcal{O}(Log\ U)\mathcal{O}(m) = \mathcal{O}(mLog\ U)$$

A su vez reemplazando en (3.5):

$$T(m,n,U) = \mathcal{O}(mLog\ U)(\mathcal{O}(m)) = \mathcal{O}(m^2 Log\ U)$$

Lo cual muestra el teorema.

3.1.3 Ventajas y desventajas

Obviamente la ventaja es la drástica disminución del tiempo de ejecución del algoritmo para valores de U grandes; si $U = 2^n$ entonces el tiempo es proporcional a $m^2 Log(2^n) = nm^2$.

Mientras que la desventaja de este algoritmo es que depende aún del valor de U, ésta dependencia la eliminaremos con el estudio de los siguientes algoritmos.

3.2 Algoritmo del Camino Aumentante más Corto (Shortest Augmenting Path)

Como su nombre lo indica éste algoritmo envía flujos a lo largo de caminos con el menor número de arcos, con ésta estrategia se logra mejorar notablemente el tiempo de ejecución del algoritmo. Para describir estos caminos se utiliza la llamada función distancia la cual pasaremos a decribir a continuación.

3.2.1 Función distancia

Consideremos una función "d" definida sobre el conjunto de nodos N, cuyos valores son enteros no negativos, llamada **función distancia**

$$\begin{array}{rccl} d: & N & \longrightarrow & \mathbb{Z}^+ \cup \{0\} \\ & i & \longrightarrow & d(i) \end{array} \tag{3.8}$$

Definición 3.2.1 (función distancia válida)

Dados una red capacitada $G = (N, A)$, dos nodos s y t llamados **origen y destino,** *definimos para cada flujo x, una* **función distancia válida** *como una función distancia que satisface:*

$$\begin{cases} d(t) = 0 \\ d(i) \leq d(j) + 1 \quad \textit{para todo arco } (i,j) \textit{ de } G(x) \end{cases}$$

Observación: Dada una función distancia válida, diremos que $d(i)$ es la **etiqueta distancia** del nodo i o simplemente la **etiqueta** del nodo i.

Al describir la función distancia válida estamos pensando en caminos que terminan en t, antes que pensar en caminos que empiezan en s.

Definición 3.2.2 *Si P es cualquier camino, definimos como la longitud de P al número de arcos que conforman dicho camino. Denotamos a la longitud de P por $\ell(P)$.*

Sean dos nodos i y j cualesquiera en la red, el camino más corto entre i y j es aquel camino cuya longitud es mínima. Es decir P^* es el camino mas corto si

$$\ell(P^*) = min\{\ell(P)/\ P \textit{ es un camino que une } i \textit{ y } j\}$$

A continuación mostramos tres propiedades que nos serán utiles al momento de mostrar la correctitud y complejidad del algoritmo "Shortest augmenting path".

Propiedad 3.2.1 *Si d es una función distancia válida respecto del flujo x, entonces para todo nodo i, cualquier camino P del nodo i hacia el nodo destino t en la red residual $G(x)$, tiene como mínimo $d(i)$ arcos*

Prueba: Sean $i \in N$ cualquiera y P cualquier camino del nodo i al nodo t en $G(x)$ cuyos nodos son $i = i_1 - i_2 - \cdots - i_k - i_{k+1} = t$. Dado que d es una función distancia válida

tenemos para cada arco del camino P:

$$\begin{aligned}
d(i_k) &\leq d(i_{k+1}) + 1 = d(t) + 1 = 0 + 1 = 1 && \text{(arco } i_k i_{k+1}\text{)} \\
d(i_{k-1}) &\leq d(i_k) + 1 \leq 1 + 1 = 2 && \text{(arco } i_{k-1} i_k\text{)} \\
d(i_{k-2}) &\leq d(i_{k-1}) + 1 \leq 2 + 1 = 3 && \text{(arco } i_{k-2} i_{k-1}\text{)} \\
&\vdots \\
d(i_2) &\leq d(i_3) + 1 \leq (k-2) + 1 = k - 1 && \text{(arco } i_2 i_3\text{)} \\
d(i) = d(i_1) &\leq d(i_2) + 1 \leq (k-1) + 1 = k && \text{(arco } i_1 i_2\text{)}
\end{aligned}$$

Por la especificación explícita del camino P, deducimos inmediatamente que su longitud está dada por k (es decir $\ell(P)$=k).

Luego entonces $d(i) \leq \ell(P) = k$ para todo nodo i y para todo camino P. Por lo tanto cualquier camino de i a t, tiene como minimo $d(i)$ arcos.

Propiedad 3.2.2 *Se cumplen las siguientes afirmaciones:*

a) Si P^* es aquel camino de i a t cuyo número de arcos es exactamente igual a $d(i)$ entonces P^* es el **camino mas corto** *entre i y t.*

b) Si $d(s) \geq n$, entonces no existe camino de s a t en $G(x)$

Prueba:

a) Esto es cierto pues, cualquier camino de i a t tiene como mínimo $d(i)$ arcos, entonces el camino con $d(i)$ arcos será el camino mas corto.

b) Por contradicción supongamos que existe un camino P de s a t en $G(x)$. P tiene como máximo $(n-1)$ arcos (caso en que $G(x)$ es un camino) y como mínimo $d(s)$ arcos (propiedad anterior), es decir:

$$d(s) \leq \ell(P) \leq n - 1$$

Lo último implica que $d(s) < n$ lo cual contradice la hipotesis.

Definición 3.2.3 (distancia exacta)

Decimos que una función distancia válida es **exacta** *si $d(i)$ representa la longitud del camino mas corto del nodo i al nodo t en la red residual $G(x)$ para cada nodo i; en otras palabras existe un camino de i a t con exactamente $d(i)$ arcos en $G(x)$.*

Definición 3.2.4 (arco admisible)

Decimos que un arco (i,j) en la red residual, es **admisible** *si satisface: $d(i) = d(j) + 1$; cualquier otro arco que no satisface ésta condición será llamado arco* **inadmisible.**

Un camino P de s a t en $G(x)$ (camino aumentante) será llamado **camino admisible,** *si todos sus arcos son admisibles.*

A continuación enunciamos la siguiente propiedad:

Propiedad 3.2.3 *Si P es un camino admisible, entonces P es el camino aumentante más corto del origen (s) al destino (t).*

Prueba: Sea P un camino admisible cuya longitud es k. Dado que cada arco (i,j) de P es admisible entonces $d(i) = d(j) + 1$, con lo cual deducimos que $d(s) = k$; por la propiedad (3.2.2(a)) P es el camino más corto de s a t en $G(x)$. Es decir P es el camino aumentante más corto.

A continuación ilustramos las definiciones anteriores y propiedades con la red residual mostrada en la figura (3.4) donde origen $s = 1$ y destino $t = 4$, la función distancia válida $d : N \longrightarrow \mathbb{Z}^+ \cup \{0\}$ está dado por: $d(1) = 2$; $d(2) = 2$; $d(3) = 1$ y $d(4) = 0$, donde evidentemente se cumple la condición $d(i) \leq d(j) + 1 \ \forall \ (i,j)$ arco de $G(x)$. Esta función distancia válida representa en particular la **distancia exacta.**

Por ejemplo $d(2) = 2$; entonces cualquier camino del nodo 2 al nodo 4 tiene como mínimo dos arcos, entre dichos nodos existen los caminos: 2-3-4 con 2 arcos y 2-1-3-4 con 3 arcos, lo mismo se puede observar entre los nodos 1 y 2.

Notemos en particular, que $d \equiv 0$ ($d(1) = d(2) = d(3) = d(4) = 0$) es también una función distancia válida pero no es exacta. Por último mencionamos que una función distancia válida exacta puede ser calculado efectuando una búsqueda por largura reversa en tiempo $\mathcal{O}(m)$

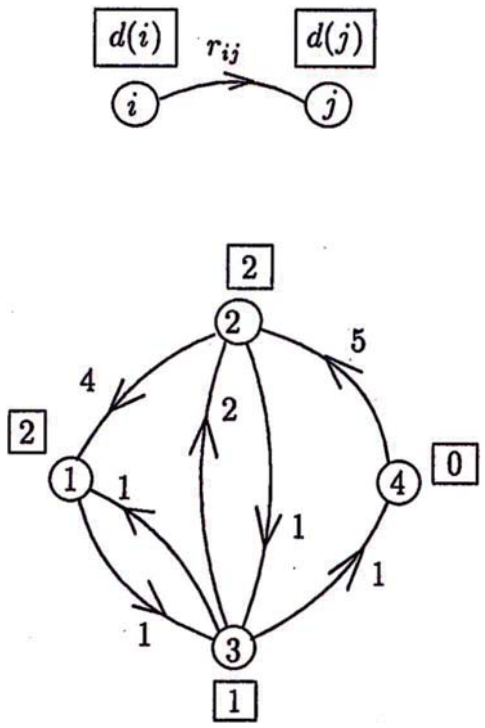


Figura 3.4: Red residual $G(x)$, con un valor $d(i)$ asociado a cada nodo

3.2.2 Implementación

El algoritmo "Shortest Augmenting Path", aumenta los flujos a través de caminos admisibles de allí el nombre del algoritmo. El camino admisible inicial es vacío y luego se incrementa el camino un arco a la vez, asi, en cada momento se tiene un camino desde el nodo origen s hacia algún nodo c al cual llamaremos **nodo actual**, el cual en algún momento llegará a ser igual al nodo destino t en cuyo caso se ha determinado un camino aumentante y por tanto se realiza la operación de **aumentación**. La función distancia se inicializa como siendo la función distancia exacta, calculada mediante una búsqueda en profundidad reversa.

El camino admisible se incrementa del siguiente modo: si el **nodo actual** c tiene un arco admisible (es decir existe $j \in A(c)$ tal que (c, j) es admisible) entonces se realiza la operación de **avance** que consiste en adicionar al camino, el arco admisible (c, j). En caso contrario, si el **nodo actual** no tiene un arco admisible entonces se realiza la operación

de **retroceso**, el cual consiste en cambiar la etiqueta del nodo actual $d(c)$ (**relabel**), de tal manera que exista al menos un arco admisible.

Realizamos todo este proceso hasta que el flujo sea máximo, el criterio de terminación está dado por $d(s) \geq n$, que asegura que el flujo es máximo (propiedad 3.2.2b)

A continuación presentamos la implementación del algoritmo:

```
r := u;
d := función exacta relativa a u;
c := s
mientras d(s) < n hacer:
    si A(c) contiene un arco admisible
       entonces        {avance}
          Sea (c,j) un arco admisible;
          pred(j) := c;
          c := j;
          si c = t
             entonces        {aumentación}
                use pred para encontrar un camino P de s a t;
                δ = min{r_ij / (i,j) es arco de P};
                para cada arco (i,j) de P hacer:
                   r_ij := r_ij − δ;
                   r_ji := r_ji + δ;
                c := s;
       sino        {retroceso}
          d(c) := min{{n} ∪ {d(j) + 1 / (c,j) ∈ A(c) y r_cj > 0}};
          si c ≠ s
             entonces
                c := pred(c)
para cada arco (i,j) de G hacer:
   x_ij := max{u_ij − r_ij, 0}
```

Figura 3.5: Algoritmo "Shortest Augmenting Path"

Ilustramos el algoritmo con el siguiente ejemplo mostrado en la figura 3.6(a), donde el nodo origen es 1 y el nodo destino es 12.

El algoritmo empieza calculando la función distancia válida exacta, utilizamos también la estructura de datos llamada "current arc" que consiste en seleccionar los arcos admisibles de la lista de adyacencia según un orden preestablecido, ésta estructura será detallada más adelante.

Tomamos como nodo actual al nodo 1, y analizamos su lista de adyacencia $A(1)$ en busca de un arco admisible, entonces elegimos el arco admisible $(1,2)$ y realizamos una operación de avance, en este momento el nodo actual es el nodo 2 y empezamos una nueva iteración, hasta este momento el camino admisible parcial es 1-2. Analizamos la lista de adyacencia $A(2)$ y seleccionamos el único arco admisible $(2,7)$ y realizamos una operación de avance, ahora el camino admisible parcial es 1-2-7, luego analizamos la lista de adyacencia $A(7)$ y elgimos el arco admisible $(7,12)$ y avanzamos, en este momento el nodo actual c es el nodo destino 12, por lo tanto esto nos indica que ha sido hallado el camino aumentante 1-2-7-12, entonces realizamos una operación de aumento enviando $\delta = min\{2,1,2\} = 1$ unidad de flujo a través de dicho camino y actualizamos las capacidades residuales de cada arco del camino aumentante (nótese que no fue necesario realizar ninguna operación de cambio de etiquetas). Hacemos el nodo actual igual al nodo origen y empezamos una nueva iteración.

Analizamos la lista $A(1)$ y elegimos el arco admisible $(1,2)$ entonces realizamos una operación de avance, entonces el nodo actual es ahora $c = 2$. Analizamos la lista de adyacencia $A(2)$ y observamos que no existe arco admisible por lo tanto es necesario realizar una operación de retroceso para ello debemos pasar por un cambio de etiqueta. Calculamos $min\{n\} \cup \{d(j) + 1 \ / \ (2,j) \in A(2), \ r_{2j} > 0\} = min\{12\} \cup \{3+1\} = 4$, entonces cambiamos la etiqueta del nodo 2 $d(2)$ que actualmente es 2, al nuevo valor 4, es decir hacemos $d(2) := 4$, hacemos el nodo actual igual al predecesor del nodo 2 es decir, nodo actual $c = 1$ y empezamos una nueva iteración, la red residual obtenida al terminar ésta iteración es mostrada en la figura 3.6(b).

Analizamos la lista de adyacencia $A(1)$ y observamos que el cambio de etiqueta en la iteración anterior hizo que el arco $(1,2)$ sea inadmisible, entonces elegimos el arco admisible $(1,3)$ y realizamos una operacion de avance, realizamos operaciones análogas y llegamos a determinar el siguiente camino admisible 1-3-8-12, a continuación cambiamos

la etiqueta del nodo 3 a $d(3) := 4$, luego se realizan los mismos pasos sobre los caminos 1-4-9-12, 1-5-10-12 y 1-6-11-12.

Analizamos la lista de adyacencia $A(1)$ y observamos que no existe arco admisible, entonces es necesario una operación de retroceso y cambio de etiqueta, calculamos: $min\{\ \{n\} \cup \{d(j)+1\ /\ (2,j) \in A(2),\ r_{2j} > 0\}\ \} = min\{\ \{12\} \cup \{4+1, 4+1, 4+1, 4+1, 4+1\}\ \} = 5$, entonces hacemos $d(1) := 5$, lo cual crea 5 arcos admisibles para el nodo 1 $(1,2)$, $(1,3)$, $(1,4)$, $(1,5)$ y $(1,6)$

Las operaciones que continuan son solo cambios de etiqueta hasta llegar a satisfacer la condicion $d(1) \geq 12$ con lo cual terminaría el algoritmo. En las siguientes iteraciones las etiquetas de los nodos 2,3,4,5,6 cambian de 4 a 5+1=6, luego la etiqueta del nodo 1 cambia de 5 a 6+1=7. A continuación las etiquetas de los nodos 2,3,4,5,6 cambian de 6 a 7+1=8 y luego la etiqueta del nodo 1 cambia de 7 a 8+1=9. En la siguiente iteración las etiquetas de los nodos 2,3,4,5,6 cambian de 8 a 9+1=10 y la etiqueta del nodo 1 cambia de 9 a 10+1=11. Se tendrá una iteración más donde la etiqueta de los nodos 2,3,4,5,6 es 12 y la etiqueta del nodo 1 es 13, como se muestra en la figura 3.7 con lo cual termina el algoritmo.

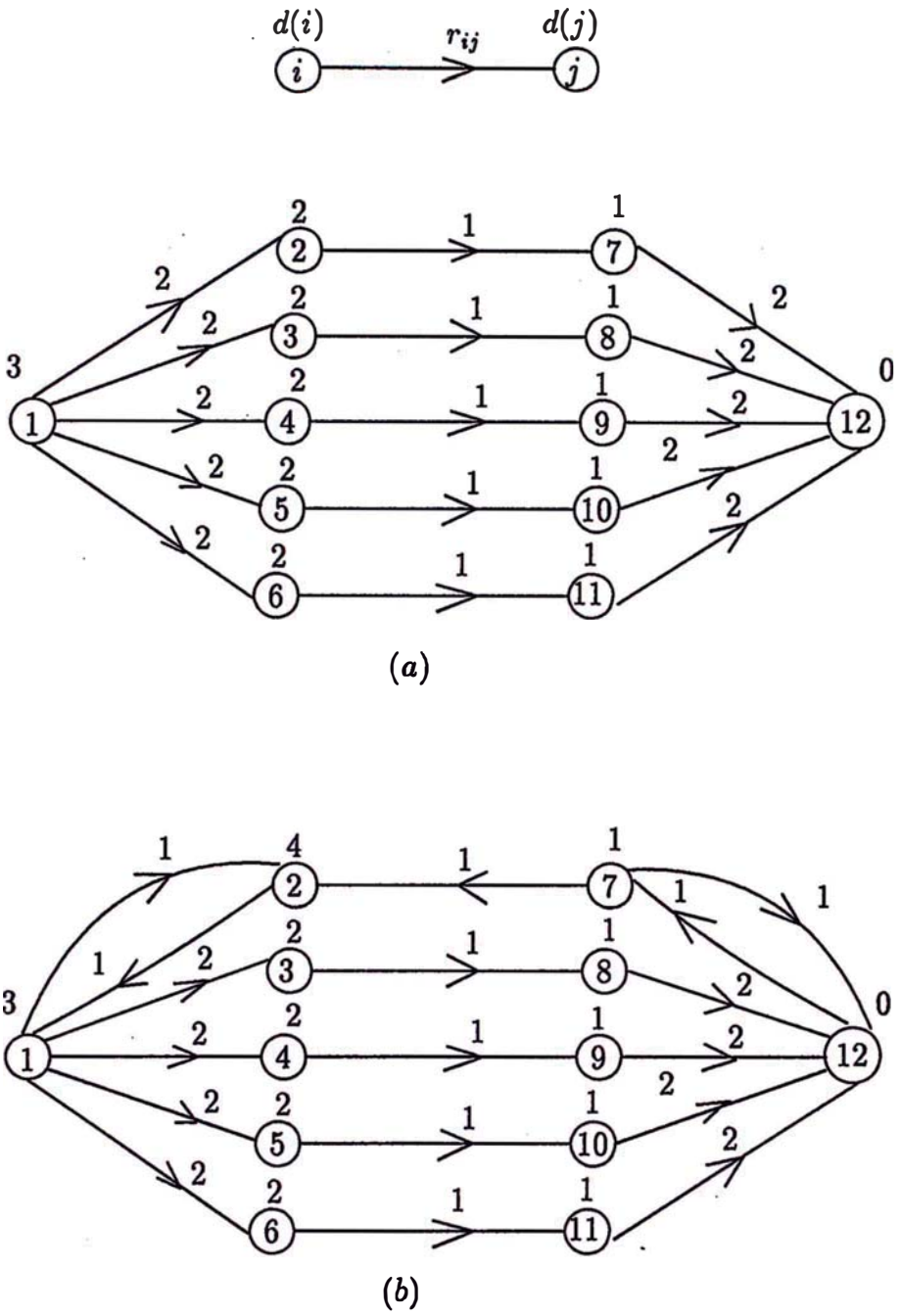


Figura 3.6: Ilustrando el algoritmo "Shortest augmenting path"; (a) red original con la función distancia válida exacta asociada a cada nodo. (b) red residual luego de enviar una unidad de flujo a través del camino 1-2-7-12 y cambiar la etiqueta del nodo 2 al nuevo valor 4

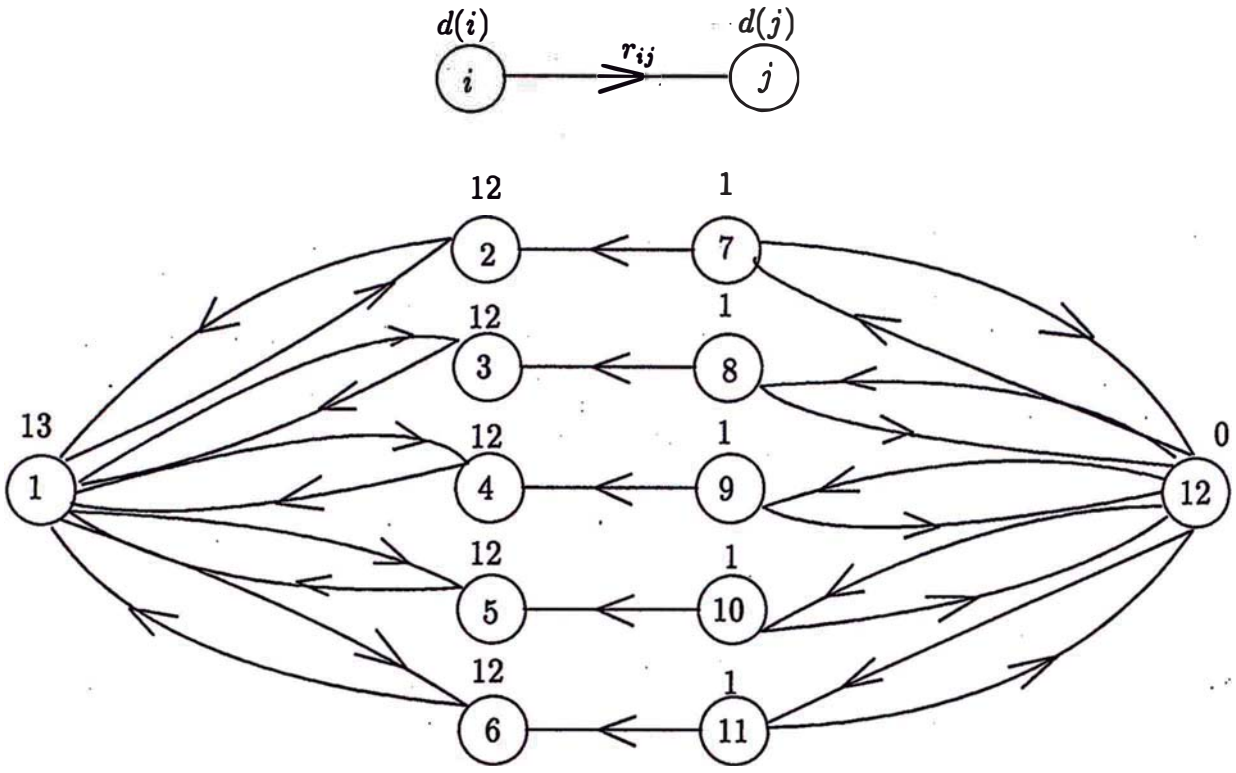


Figura 3.7: Red residual obtenida después de la última iteración, todas las capacidades residuales son iguales a la unidad ($r_{ij} = 1 \; \forall \; i, \; j$)

3.2.3 Correctitud y complejidad

correctitud

Para demostrar la correctitud del algoritmo necesitamos probar el siguiente lema:

Lema 3.2.1 *El algoritmo "Shortest Augmenting Path", mantiene la función distancia como válida en cada paso. Además si $d(i)$ es la etiqueta del nodo i, y se realiza una operación de* **retroceso o relabel**, *entonces la nueva etiqueta $d'(i)$ aumenta estrictamente es decir, $d'(i) > d(i)$*

Prueba: Mostramos que la función distancia o las etiquetas se mantienen como válidas en cada paso, realizando inducción en el número de **aumentaciones** y el número de operaciones de **retroceso** (relabels o cambios de etiqueta). La operación de **avance** no

afecta la admisibilidad de ningun arco; pues no cambia ni la capacidad residual ni mucho menos las etiquetas, entonces el número de operaciones de avance no se toma en cuenta para realizar la inducción.

Inicialmente el algoritmo construye la función distancia exacta, por lo tanto todas las etiquetas son válidas, Asumimos como hipótesis de inducción que las etiquetas son válidas antes de realizar una operación (aumentación o retroceso). Debemos mostrar que las nuevas etiquetas continúan siendo válidas luego de la operación correspondiente (aumentación o retroceso).

a) Si la operación fue aumentación, quiere decir que se ha logrado establecer un camino aumentante, sea P tal camino, sea además (i,j) cualquier arco de P.

Si la aumentación satura el arco (i,j), es decir el arco (i,j) desaparece de la red residual, entonces las etiquetas permanecen válidas, pues simplemente está desapareciendo un arco.

Por otro lado la aumentación puede crear el nuevo arco reverso (j,i) en la red residual, y por lo tanto se debe verificar la validéz de las etiquetas sobre los extremos $d(j)$ y $d(i)$ de este arco, (es decir debemos verificar que se cumple $d(j) \leq d(i)+1$); para ello consideremos como dato la admisibilidad del arco (i,j), entonces se tiene $d(i) = d(j)+1$; luego $d(j) = d(i)-1 < d(i)+1 \leq d(i)+1$, por lo tanto tenemos que $d(j) \leq d(i)+1$; es decir las etiquetas de los extremos del nuevo arco continúan siendo válidas. Como el análisis anterior fue realizado para cualquier arco (i,j) del camino aumentante P y los posibles cambios solo pueden ocurrir a lo largo del camino P, entonces la función distancia permanece válida después de la aumentación.

b) Si la operación fue **retroceso o relabel**, sea c el **nodo actual**, para el cual no existe arco admisible en $A(c)$ ($\nexists\ j \in A(c)$ tal que $d(c) = d(j)+1$). En este caso el algoritmo cambia la etiqueta del nodo c, sea $d'(c)$ la nueva etiqueta. Entonces para probar que las etiquetas continúan válidas se debe mostrar que los arcos que **salen** de c y los arcos que **entran** hacia c, continúan siendo válidas con respecto a la nueva etiqueta $d'(c)$ del nodo c.

para arcos que salen de c

Por hipótesis de inducción se cumple $d(i) \leq d(j) + 1$ para todo arco (i, j) en la red residual. Al no existir ningún arco admisible para c en $A(c)$ (ningún arco (c, j) en A(c) cumple: $d(c) = d(j) + 1$), se tiene que:

$$d(c) < d(j) + 1 \quad \forall\, (c, j) \in A(c) \tag{3.9}$$

Además el algoritmo calcula una nueva etiqueta para c; dado por:

$$d'(c) = min\{d(j) + 1 \;/\; (c, j) \in A(c) \text{ y } r_{cj} > 0\} \tag{3.10}$$

Como

$$min\{d(j) + 1 \;/\; (c, j) \in A(c) \text{ y } r_{cj} > 0\} \leq d(j) + 1 \quad \forall\, (c, j) \in A(c)$$

entonces utilizando este resultado conjuntamente con (3.10) obtenemos:

$$d'(c) \leq d(j) + 1 \quad \forall\, (c, j) \in A(c)$$

lo cual muestra que todos los arcos que salen del nodo c continuan siendo válidas después del cambio de etiqueta.

para arcos que entran hacia c

De la ecuación (3.9) podemos afirmar:

$$d(c) < min\{d(j) + 1 \;/\; (c, j) \in A(c) \text{ y } r_{cj} > 0\} = d'(c) \tag{3.11}$$

es decir

$$d'(c) > d(c) \tag{3.12}$$

lo cual muestra que la etiqueta de c aumenta su valor estrictamente luego de la operación de **relabel** (cambio de etiqueta).

Veamos ahora que ocurre con los arcos que entran hacia el nodo c. Sea (k, c) cualquier arco que entra hacia el nodo c, por hipótesis de inducción las etiquetas son válidas entonces tenemos que:

$$d(k) \leq d(c) + 1$$

Utilizando la ecuación (3.12) afirmamos:

$$d(k) \leq d(c) + 1 < d'(c) + 1$$

es decir,

$$d(k) \leq d'(c) + 1$$

Con lo cual los arcos que entran hacia el nodo c, continúan siendo válidas luego del cambio de etiqueta del nodo c

Todo el argumento anterior sirve para demostrar la correctitud del algoritmo, es decir el algoritmo termina y cuando ello ocurre devuelve el flujo máximo.

Dado que el algoritmo inicializa $d(s)$ como la longitud del camino mas corto del origen al destino entonces el valor de $d(s)$ es estrictamente menor que n, además los argumentos anteriores muestran que al cambiar el valor de $d(s)$ los nuevos valores son estrictamente mayores que los anteriores. Por lo tanto se asegura que el valor de $d(s)$ llegará a ser el algún momento igual a n con lo cual el algoritmo termina.

Además si el algoritmo finalizó entonces se tiene que $d(s) = n$, luego por la propiedad (3.2.2(b)), no existe camino aumentante en la red residual $G(x)$ y finalmente por el teorema (2.7) del capítulo 2 afirmamos que el flujo x es máximo. Lo cual termina de probar la correctitud del algoritmo

complejidad

A continuación, estudiemos cuánto tiempo demora el algoritmo en resolver cualquier problema de flujo máximo, para ello necesitamos dar y mostrar algunas definiciones y lemas.

El algoritmo siempre examina los arcos de la lista de adyacencia del nodo actual c; si la implementación se realiza sin mucho cuidado, entonces cada arco será examinado un número excesivo de veces, y por consiguiente el tiempo de ejecución del algoritmo será demasiado alto. Entonces queda claro que es necesario establecer algun patrón para examinar los arcos de $A(c)$ a lo largo de las iteraciones, de un modo eficiente.

La estructura de datos que utilizaremos para este fin se denomina **current arc**, que consiste en lo siguiente: Dada la lista de adyacencia $A(c)$, ordenamos los arcos de dicha

lista de un modo arbitrario pero fijo, este orden se mantendrá a lo largo de la ejecución del algoritmo. Entonces para examinar los arcos de $A(c)$ lo haremos de acuerdo al orden establecido, el **current arc** es el primer arco candidato de la lista $A(c)$, para ser verificado si satisface o no la admisibilidad. Entonces siempre que el algoritmo necesita de un arco admisible se fija en el **current arc**; si es admisible lo toma y designa como nuevo **current arc** al siguiente arco de la lista $A(c)$. Si por el contrario no es admisible entonces también designa como nuevo **current arc** al siguiente arco de la lista y asi hasta encontrar un arco admisible o en su defecto hasta determinar que la lista $A(c)$ no tiene arco admisible.

En el caso que $A(c)$ no tiene arco admisible, la situación continúa así hasta que se realize una operación de relabel (o cambio de etiquetas), formalizamos esta observación en el siguiente lema.

Lema 3.2.2 *Si un arco (i,j) es inadmisible en cierta iteración del algoritmo, entonces continuará siendo inadmisible hasta que la etiqueta del nodo i $d(i)$ cambie, (según el lema (3.2.1) dicho cambio se refleja en un aumento estricto de valor).*

Prueba: Se supone que en cada momento del algoritmo la función d es válida, entonces $d(i) \leq d(j)+1$, como el arco (i,j) es inadmisible entonces no se cumple la igualdad, luego:

$$d(i) < d(j) + 1 \tag{3.13}$$

Si la etiqueta del nodo j $d(j)$ cambia, esto no efecta a la inadmisibilidad del arco (i,j), pues si $d'(j)$ es la nueva etiqueta de j, entonces se tiene que $d(j) < d'(j)$ (por el lema (3.2.1); por lo tanto reemplazando en (3.13) tenemos que: $d(i) < d'(j) + 1$, con lo cual el arco (i,j) continúa siendo inadmisible, después de un cambio de etiquetas del nodo j. Entonces el estado de inadmisibilidad del arco (i,j) solo puede cambiar con el cambio de etiqueta en el nodo i, caso que veremos a continuación.

Para mostrar que el lema es cierto procedamos por contradicción, supongamos que el arco (i,j) se convierte en admisible, antes de un cambio de etiqueta en el nodo i, entonces el que debe cambiar es la etiqueta del nodo j, sea $d'(j)$ la nueva etiqueta del nodo j. Entonces se debe tener que:

$$d(i) = d'(j) + 1 \tag{3.14}$$

por el lema (3.2.1) se cumple que $d(j) < d'(j)$, entonces reemplazando en la ecuación (3.14) tenemos que :

$$d(i) = d'(j) + 1 > d(j) + 1$$

es decir,

$$d(i) > d(j) + 1$$

lo cual contradice a la ecuación (3.13).

El lema anterior nos permite establecer la siguiente propiedad:

Propiedad 3.2.4 *Si el algoritmo cambia la etiqueta de algún nodo a lo mas "k" veces, entonces el tiempo que gasta en encontrar arcos admisibles y por consiguiente cambiar la etiqueta de los nodos está dado por:*

$$\mathcal{O}\left(k\sum_{i\in N}|A(i)|\right) = \mathcal{O}(km)$$

A continuación, mostramos los siguientes lemas, que nos permitiran establecer una cota superior para el tiempo de ejecución del algoritmo.

Lema 3.2.3 *Si el algoritmo cambia la etiqueta de algún nodo a lo más "k" veces, entonces satura arcos a lo más* $\frac{km}{2}$ *veces.*

Prueba: ver [3]

Lema 3.2.4 *En el algoritmo "Shortest augmenting path" se cumplen las siguientes afirmaciones:*

a) Cada etiqueta incrementa su valor a lo más n veces

b) El número total de cambios de etiqueta es a lo más n^2

c) El número de aumentaciones es a lo más $\frac{mn}{2}$ *veces*

Prueba:

a) Supongamos que la etiqueta de algún nodo i cambió k veces, sean dichas etiquetas: $d(i) < d'(i) < d''(i) < d^{(3)}(i) < \cdots < d^{(k-1)})(i) < d^{(k)}(i)$, entre dos cambios consecutivos de etiqueta, el valor de la misma incrementa su valor en al menos 1 unidad, es decir por ejemplo:

$$d'(i) \geq d(i) + 1\ , \quad d''(i) \geq d'(i) + 1\ , \quad d'''(i) \geq d''(i) + 1\ , \quad \text{etc.}$$

de lo cual podemos establecer:

$$d(i) + k \leq d^{(k)}(i)$$

Además sabemos que, $d^{(k)}(i) < n$ (pues de lo contrario el algoritmo ya habría terminado). Por lo tanto tenemos que:

$$d(i) + k \leq d^{(k)} < n\ ,$$

dado que $d(i) + k \geq k$, entonces concluimos que:

$$k < n$$

Es decir el número de veces que la etiqueta de cualquier nodo i, sufre modificación (en consecuencia aumenta), es como máximo n. En otras palabras, la etiqueta de cualquier nodo, se incrementa a lo más n veces.

b) Es inmediato a partir de *a*) tenemos n nodos y cada nodo cambia de etiqueta como máximo n veces, en consecuencia el número total de cambios de etiqueta es como máximo n^2.

c) Por la parte *a*) el algoritmo cambia la etiqueta de cualquier nodo a lo mas n veces, entonces por el lema (3.2.3), el número de aumentaciones es a lo más $\frac{mn}{2}$

Teorema 3.3 *La complejidad de tiempo del algoritmo "Shortest augmenting path" es* $\bullet(mn^2)$

Prueba: La complejidad de tiempo del algoritmo, está dado por la suma de las complejidades de las operaciones involucradas; cada una de las cuales pasamos a detallar.

Por el lema 3.2.4(c), el algoritmo realiza $\mathcal{O}(mn)$ operaciones de aumentación, además cada aumentación requiere un tiempo proporcional a n, en consecuencia:

$$\text{tiempo gastado en aumentaciones} = \mathcal{O}(mn)\mathcal{O}(n) = \mathcal{O}(mn^2) \tag{3.15}$$

Por otra parte el algoritmo, realiza a lo más n^2 operaciones de retroceso (cambios de etiqueta), y dado que cada cambio de etiqueta se realiza en tiempo constante entonces:

$$\text{tiempo gastado en retrocesos} = \mathcal{O}(n^2) \tag{3.16}$$

A continuación veamos cuánto tiempo gasta el algoritmo en realizar las operaciones de avance, cada operación de avance adiciona un arco al camino admisible parcial, por lo tanto el número de operaciones de avance está limitado por n (longitud máxima de un camino admisible), inmediatamente después ocurrirá o bien una operación de **retroceso** o bien una operacion de **aumentación**, entonces es claro que el tiempo gastado en realizar las operaciones de avance depende del tiempo gastado en aumentaciones y retrocesos. En consecuencia la estimativa mas desfavorable para el tiempo total gastado en operaciones de avance es (considerando 3.15 y 3.16):

tiempo gastado en avances = tiempo gastado en retrocesos+tiempo gastado en aumentacion

es decir:

$$\text{tiempo gastado en avances} = \mathcal{O}(n^2) + \mathcal{O}(mn^2) = \mathcal{O}(n^2 + mn^2) \tag{3.17}$$

De las ecuaciones (3.15, 3.16 y 3.17) se concluye que la complejidad de tiempo $T(m,n)$ del algoritmo "Shortest augmenting path" es:

$$\begin{aligned}
T(m,n) &= T_{aumentaciones} + T_{retrocesos} + T_{avances} \\
&= \mathcal{O}(mn^2) + \mathcal{O}(n^2) + \mathcal{O}(n^2 + mn^2) \\
&= \mathcal{O}(mn^2 + n^2 + n^2 + mn^2) = \mathcal{O}(n^2 + mn^2) \\
&= \mathcal{O}(n^2(m+1)) \\
&= \mathcal{O}(n^2)\mathcal{O}(m+1) \\
&= \mathcal{O}(n^2)\mathcal{O}(m) \\
&= \mathcal{O}(mn^2)
\end{aligned}$$

Es decir $T(m,n) = \mathcal{O}(mn^2)$, lo cual demuestra el teorema.

3.2.4 Ventajas y desventajas

Obviamente la gran ventaja del algoritmo es su complejidad polinomial y su no dependencia del número U, con respecto al algoritmo anterior ("Capacity scaling").

Lamentablemente la implementación particular que estamos estudiando aún tiene una desventaja de tipo práctica, pero dicha desventaja es fácilmente superable adicionando algunos detalles al algoritmo, estos detalles son estudiados a continuación.

El criterio de terminación del algoritmo ($d(s) \geq n$), es satisfactorio, pero no es eficiente en la práctica, investigaciones empíricas han demostrado que el algoritmo gasta mucho tiempo en las últimas iteraciones, cambiando la etiqueta de los nodos hasta llegar a la condición de terminación $d(s) \geq n$, cuando el flujo máximo ya ha sido hallado por el algoritmo, pero el algoritmo no es capáz de detectar que el flujo máximo ya ha sido hallado.

Introducimos una variante en el algoritmo, gracias a la cual se detectará la presencia del corte mínimo y en consecuencia la del flujo máximo, mucho antes que la etiqueta de nodo s satisfaga la condición $d(s) \geq n$.

Ilustramos esta variante, con el mismo ejemplo mostrado en la figura (3.6(a)), en dicho ejemplo se observa que el algoritmo envía una unidad de flujo a través de los caminos 1-2-7-12, 1-3-8-12, 1-4-9-12, 1-5-10-12 y 1-6-11-12 y cambia la etiqueta de los nodos 2,3,4,5 y 6 de 2 al nuevo valor 4, en este momento el flujo máximo ya ha sido hallado pues ya no es posible realizar ninguna aumentación, sin embargo el algoritmo aún no termina, debido a que la condición de terminación ($d(1) \geq 12$) aún no se satisface. Es por eso que el algoritmo cambia la etiqueta de los nodos 1 2 3 4 5 y 6, repetitivamente hasta alcanzar la condición $d(1) \geq 12$ sin realizar ninguna aumentación, estos cambios de etiqueta innecesarios son los que evitaremos con la siguiente variante del algoritmo (esta se debe a [6]).

Consideremos la situación inmediatamente después de enviar 1 unidad de flujo a través de los caminos 1-2-7-12, 1-3-8-12, 1-4-9-12, 1-5-10-12 y 1-6-11-12 y cambiar la etiqueta de los nodos 2,3,4 y 5, esta situación es mostrada en la figura (3.8).

Sea la función *num*:

$$\begin{array}{rccl} num: & \{0,1,2,\cdots,(n-1)\} & \longrightarrow & \{0,1,2,\cdots,n\} \\ & k & \longrightarrow & num(k) \end{array} \tag{3.18}$$

donde, $num(k)$ es el número de nodos cuya etiqueta es igual a k,

Esta función es inicializada al momento de calcular la función distancia válida exacta, al inicio del algoritmo y su valor varía cada vez que se realiza un cambio de etiqueta. Si en cierto momento del algoritmo algún nodo k cambia su etiqueta de k_1 al nuevo valor k_2, (es decir, $d(k) = k_1$ y luego del cambio de etiqueta $d(k) = k_2$), entonces el valor de $num(k_1)$ disminuye en 1 y el valor de $num(k_2)$ aumenta en 1.

Si la condición $num(k_1) = 0$ se cumple, entonces el algoritmo termina, por ejemplo en la figura (3.8), la función *num* está dada por: $num(0) = 1$, $num(1) = 5$, $num(2) = 1$, $num(3) = 1$, $num(4) = 4$ y $num(i) = 0$ para todo $i = 5, 6, \cdots 11$, en la siguiente iteración el algoritmo elige el camino admisible parcial 1-6 y al no existir otro arco admisible, la etiqueta del nodo $k = 6$ cambia de $k_1 = 2$ al nuevo valor $k_2 = 4$, en consecuencia $num(2) := num(2)-1 = 1-1 = 0$ y $num(4) := num(4)+1 = 4+1 = 5$, es decir $num(k_1) = 0$ y $num(k_2) = 5$, dado que $num(k_1) = 0$ entonces el algoritmo termina, evitándose iteraciones (cambios de etiqueta) innecesarios. Pues los conjuntos $S = \{1,2,3,4,5,6\}$ y $\overline{S} = \{7,8,9,10,11,12\}$ determinan un corte $[S, \overline{S}]$ que separan $s = 1$ y $t = 12$, y además dicho corte es mínimo, en consecuencia ya estamos ante la presencia del flujo máximo. Todas estas razones son justificadas a continuación.

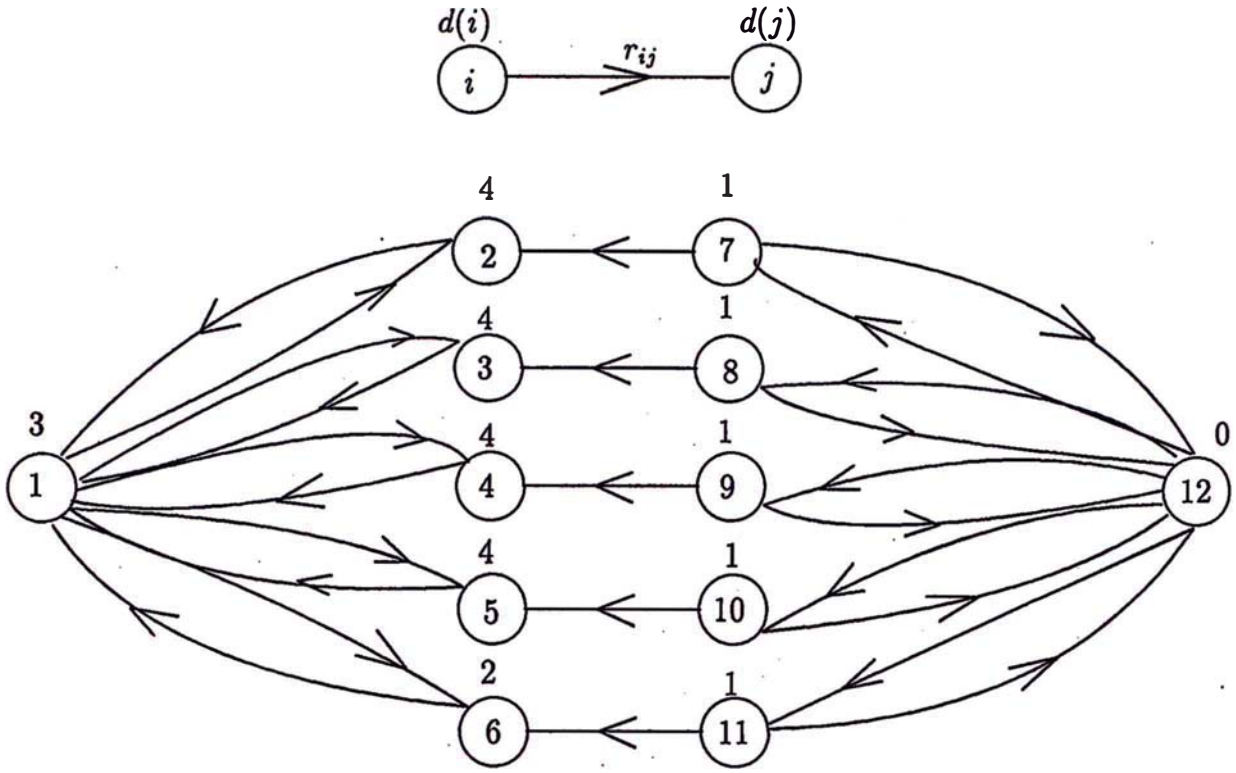


Figura 3.8: Mejorando el tiempo de ejecución del algoritmo del camino aumentante mas corto $r_{ij} = 1 \ \forall \ i, j$

Para mostrar la correctitud de esta variante consideremos los conjuntos:

$$S = \{i \in N \ / \ d(i) > k_1\} \qquad \text{y} \qquad \overline{S} = \{i \in N \ / \ d(i) < k_1\}$$

Propiedad 3.2.5 $[S, \overline{S}]$ *es un corte que separa s y t cuya capacidad es mínima*

Prueba: Sea k el nodo cuya etiqueta es k_1 (es decir, $d(k) = k_1$), es claro que existe un camino del nodo s al nodo k, pues el algoritmo está construyendo un camino admisible parcial desde el nodo origen s, por la admisibilidad del camino tenemos que $d(s) > d(k)$, además como $k \neq t$, entonces $d(k) > 0$, también $d(t) = 0$, por lo tanto: $d(s) > d(k) > d(t)$, es decir $d(s) > k_1 > d(t)$, la última relación conjuntamente con las definiciones de S y $\overline{S}$, demuestran que $s \in S$ y $t \in \overline{S}$, en consecuencia $[S, \overline{S}]$ es un corte que separa s y t.

Veamos ahora que dicho corte es mínimo, sea $(i, j) \in [S, \overline{S}]$ cualquier arco del corte, entonces $i \in S$ y $j \in \overline{S}$, y por lo tanto $d(j) < k_1 < d(i)$, equivalentemente $d(j) + 1 \leq k_1 <$

$d(i)$, entonces $d(j) + 1 < d(i)$, es decir:

$$d(i) > d(j) + 1 \qquad \forall (i,j) \in [S, \overline{S}]$$

en consecuencia, $r_{ij} = 0$ para todo $(i,j) \in (S, \overline{S})$, pues si algún arco satisface $r_{ij} > 0$, entonces se debe cumplir $d(i) \leq d(j) + 1$.

Como $r_{ij} = 0$ para todo $(i,j) \in (S, \overline{S})$, entonces por el corolario (2.4), $[S, \overline{S}]$ es un corte de capacidad mínima, lo cual demuestra la propiedad.

Es claro que la propiedad anterior, es la demostración de la correctitud de ésta variante del algoritmo, es decir que el flujo obtenido ya es máximo.

Otra ventaja del algoritmo "Shortest augmenting path", es que puede ser adaptado al algoritmo "Capacity Scaling" (estudiado en la seccion anterior), con el objeto de reducir su complejidad de tiempo. Recordemos que la complejidad de tiempo del algoritmo "Capacity scaling" fue estimado como siendo $\mathcal{O}(m^2 Log(U))$, este tiempo puede ser reducido a $\mathcal{O}(mnLog(U))$, utilizando la estrategia del camino aumentante mas corto.

3.3 Algoritmo "Preflow Push"

Estudiaremos a continuación un tipo diferente de algoritmo a todos los anteriores, para el problema del flujo máximo conocido como "Preflow push". Estudiaremos el funcionamiento del algoritmo, sin dar muchos detalles acerca de las pruebas de correctitud y complejidad del algoritmo, sin embargo daremos la referencia donde se pueden encontrar dichas pruebas.

La característica fundamental de los algoritmos anteriores es construir un camino aumentante de alguna forma u otra, y realizar la aumentación a través de dichos caminos, además en cada paso del algoritmo se dispone de un flujo cuyo valor se incrementa con cada aumentación. Esta propiedad (la de construir un camino aumentante) hace que los algoritmos de caminos aumentantes, tengan la desventaja inherente de demorar un tiempo $\mathcal{O}(n)$ en el peor caso, para enviar δ unidades de flujo a través del camino aumentante hallado.

A diferencia de los algoritmos de caminos aumentantes, el algoritmo "Preflow push",

no dispone de un flujo en los pasos intermedios, si no mas bien dispone de una función denominada **preflujo** que definiremos mas adelante. El flujo solo es calculado al terminar el algoritmo, además el algoritmo "Preflow push" envía los flujos a través de arcos y no a través de caminos aumentantes, mientras que los algoritmos de caminos aumentantes envían flujo a través de un solo camino en cada paso, el algoritmo "Preflow push", envía flujo a través de varios caminos a la vez.

Es por la última razón que éste algoritmo es mejor que los algoritmos de caminos aumentantes, tanto en la teoría como en la práctica. Nosotros describimos el algoritmo genérico de ésta clase de algoritmos y luego mencionamos implementaciones específicas.

3.3.1 Preflujos

Definición 3.3.1 (Preflujo)

Un Preflujo es una función x:

$$\begin{aligned} x: \quad A &\longrightarrow \mathbb{Z} \\ (i,j) &\longrightarrow x(i,j) := x_{ij} \end{aligned}$$

tal que:

$$\sum_{\{j \,/\, (i,j)\in A\}} x_{ji} - \sum_{\{j \,/\, (i,j)\in A\}} x_{ij} \geq 0 \quad \forall \; i \in N \setminus \{s,t\} \tag{3.19}$$

$$x_{ij} \leq u_{ij} \quad \forall \; (i,j) \in A \tag{3.20}$$

La ecuación (3.19) es una relajación de la condición de **equilibrio de masa**, dada en la definición de flujo, y nos dice que **lo que entra en i menos lo que sale de i es mayor o igual a cero**, mientras que la ecuación (3.20) nos dice que el preflujo x **respeta** u.

Definición 3.3.2 (Exceso)

Dado un preflujo x, definimos para cada nodo i, el exceso del nodo i, al número:

$$e(i) = \sum_{\{j \,/\, (i,j)\in A\}} x_{ji} - \sum_{\{j \,/\, (i,j)\in A\}} x_{ij}$$

Es claro que $e(i) \geq 0 \; \forall i \in N \setminus \{s,t\}$.

Decimos que un nodo i es **activo** si $e(i) > 0$ (es decir, si su exceso es estrictamente positivo), adoptamos como convención, que el nodo origen s y el nodo destino t, nunca son activos.

En los algoritmos de caminos aumentantes siempre tenemos una solución admisible, la cual será optima al finalizar el algoritmo, sin embargo en el algoritmo "Preflow push" la presencia de nodos activos, nos indica que la solucion aún es inadmisible, cuando ya no existen nodos activos, entonces la solución se hace admisible y mas aún óptima. Por lo tanto es prioridad en el algoritmo seleccionar un nodo activo y disminuir su exceso hasta llegar a eliminarlo.

Para ello, luego de seleccionar un nodo activo enviamos flujo hacia sus nodos vecinos, vale decir a través de los arcos que se encuentran en la lista de adyacencia del nodo activo seleccionado, pero no a todos ellos, si no a aquellos nodos que se encuentran más **cerca** del nodo destino t, como en el algoritmo del camino aumentante más corto, medimos esta cercanía mediante la función distancia válida, es decir elegido el nodo activo, enviamos flujo a través de los arcos de la lista de adyacencia que son admisibles.

3.3.2 Implementación

El algoritmo funciona del siguiente modo: se calcula la función distancia válida exacta inicial, luego se saturan todos los arcos de $A(s)$, consecuentemente hacemos $d(s) = n$, a continuación elegimos el nodo activo i, y luego enviamos flujo a través del arco $(i, j) \in A(i)$, si y solamente si (i, j) es admisible, es decir, si y solamente si $d(i) = d(j)+1$. continuamos este proceso hasta que no existan nodos activos, con lo cual se obtiene el flujo máximo.

Escribimos el algoritmo, de la manera más eficiente, es decir, utilizando la estructura de datos **current arc**.

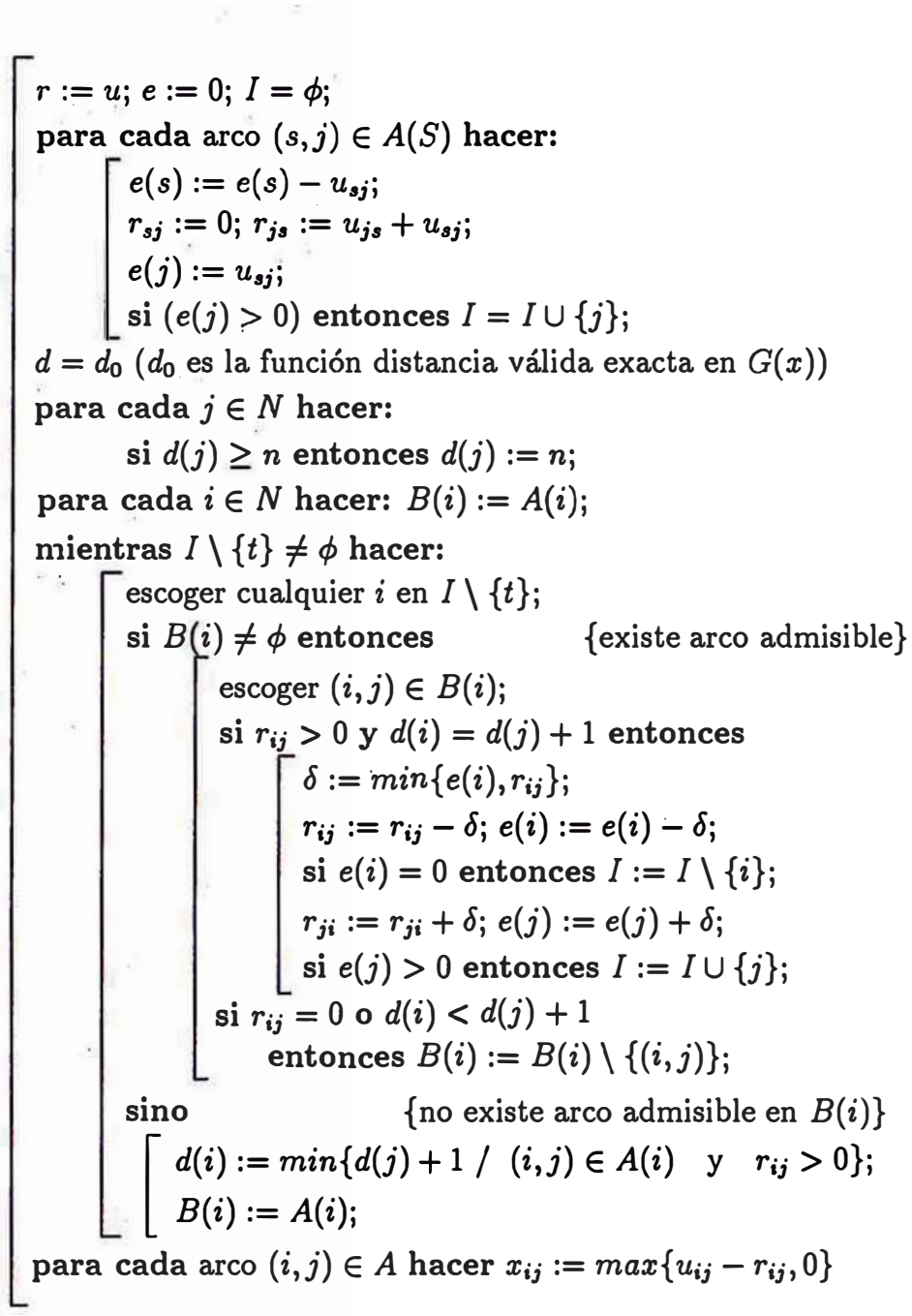


Figura 3.9: Algoritmo "Preflow Push", con current arc

Aquí I es el conjunto de nodos activos, y $B(i)$ es el conjunto de nodos admisibles para cada i.

Ilustramos el algoritmo anterior con el problema del flujo máximo dado en la figura 3.10(a), donde el nodo origen es $s = 1$ y el nodo destino es $t = 6$.

El primer paso consiste en realizar un preproceso de los datos iniciales, es decir calcular

las distancias exactas, inicializar los excesos y enviar el máximo flujo posible a través de los arcos adyacentes al nodo origen, esta última operación modifica los excesos del nodo origen y el de sus adyacentes. El resultado del preproceso es mostrado en la figura 3.10(b).

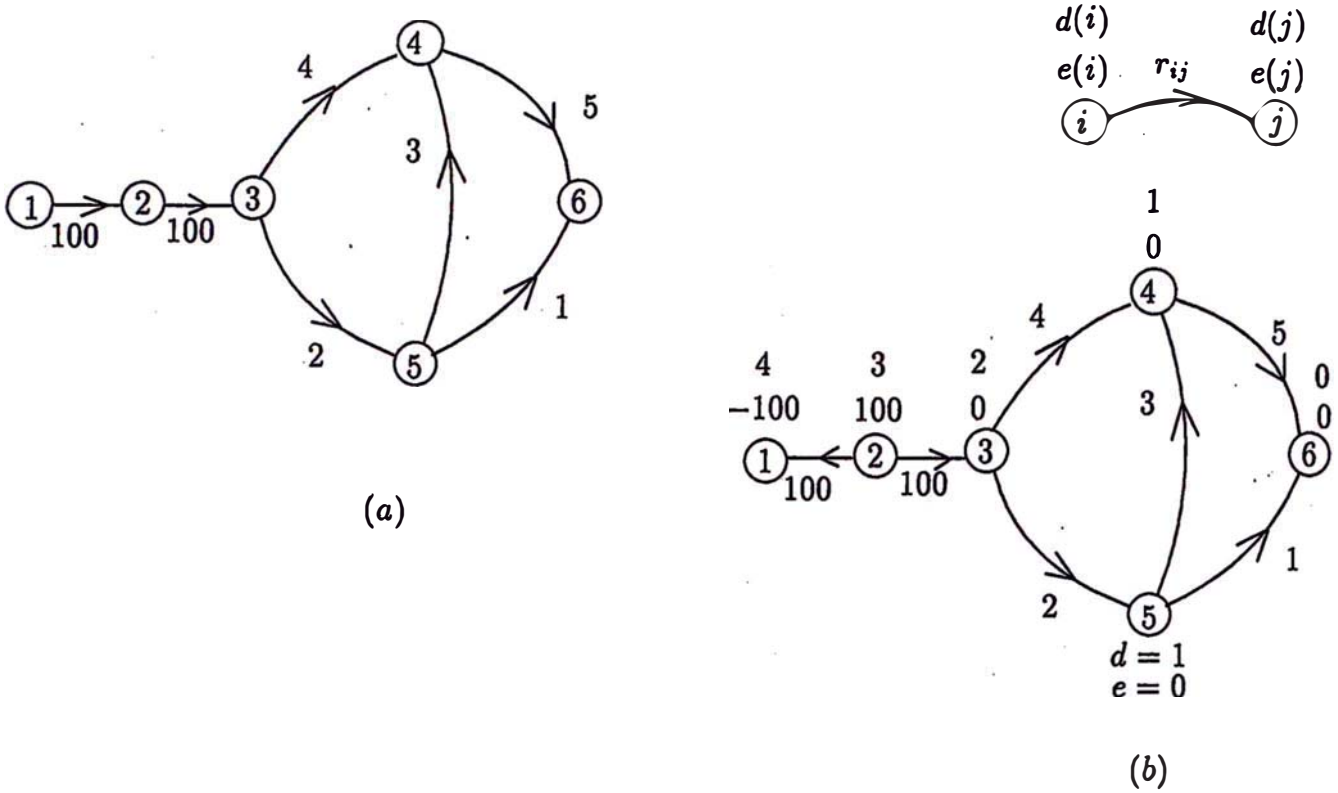


Figura 3.10: (a) Red inicial; (b) Red residual luego del preproceso

Las iteraciones del algoritmo son las siguientes:

Primera iteración: Supongamos que el algoritmo selecciona el nodo activo $i = 2$, este nodo tiene un exceso $e(2) = 100$ (como se muestra en la figura 3.10(b)) y además tiene un solo arco admisible $(2, 3)$, entonces calculamos $\delta = min\{e(2), r_{23}\} = min\{100, 100\} = 100$, y enviamos o empujamos (**push**) $\delta = 100$ unidades de flujo a través del arco $(2, 3)$, con lo cual el exceso del nodo 2 disminuye en $\delta = 100$ unidades, es decir $e(2) = 100 - 100 = 0$ y el exceso del nodo 3 aumenta en 100 unidades, es decir $e(3) = 0 + 100 = 100$, además las capacidades residuales sobre los arcos (2,3) y

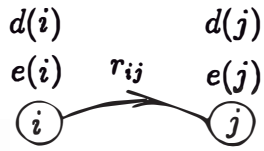
(3,2) se actualizan de la misma forma que en los algoritmos anteriores ($r_{23} := r_{23} - \delta$ y $r_{32} := r_{32} - \delta$).

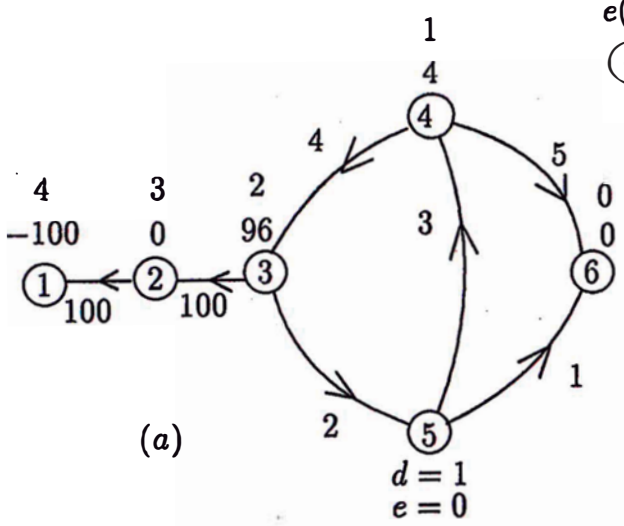
Segunda iteración: Supongamos que el algoritmo ahora selecciona el nodo activo $i = 3$, este nodo tiene un exceso $e(3) = 100$ y dos arcos admisibles, elegimos el arco $(3,4)$ y calculamos $\delta = min\{e(3), r_{34}\} = min\{100, 4\} = 4$, a continuación enviamos o empujamos (**push**) $\delta = 4$ unidades de flujo a través del arco admisible $(3,4)$, como consecuencia el exceso del nodo 3 disminuye en $\delta = 4$ unidades ($e(3) = 100 - 4 = 96$) y el exceso del nodo 4 aumenta en 4 unidades ($e(4) = 0 + 4$), también las capacidades residuales son actualizados; la red residual resultante es mostrado en la figura 3.11(a).

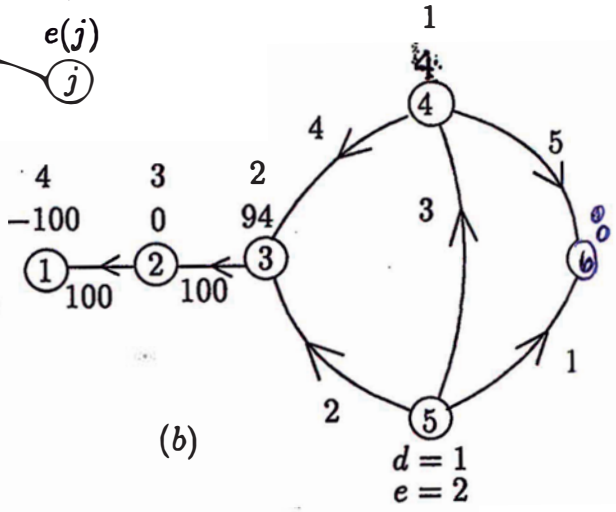
Tercera iteración: Elegimos nuevamente el nodo activo $i = 3$ de la red que se muestra en la figura 3.11(a), el cual tiene un exceso $e(3) = 96$ y el arco admisible $(3,5)$, entonces calculamos $\delta = min\{e(3), r_{35}\} = min\{96, 2\} = 2$, luego enviamos $\delta = 2$ unidades de flujo a través del arco $(3,5)$ con lo cual el exceso del nodo 3 disminuye en 2 unidades ($e(3) = 96 - 2 = 94$)y el exceso del nodo 5 aumenta en 2 unidades ($e(5) = 0 + 2 = 2$). La red residual resultante se muestra en la figura 3.11(b).

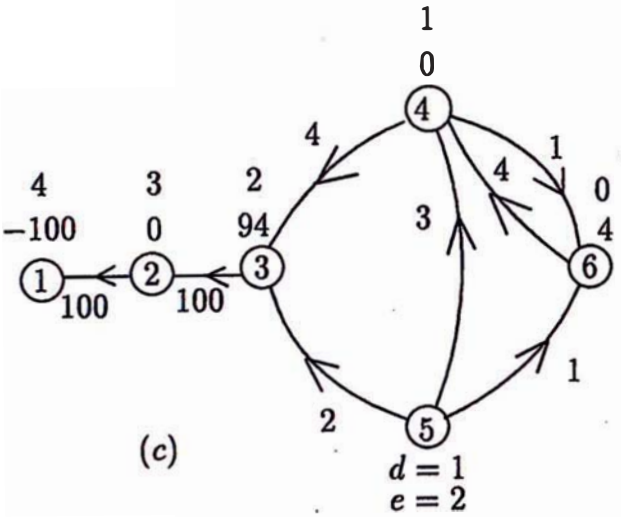
Cuarta iteración: Se elige el nodo activo $i = 4$, cuyo exceso es 4, consideramos el arco admisible $(4,6)$ y calculamos $\delta = min\{4, 5\} = 4$, por lo tanto se envía $\delta = 4$ unidades de flujo a través del arco $(4,6)$, obteniendo la red residual mostrada en la figura 3.11(c)

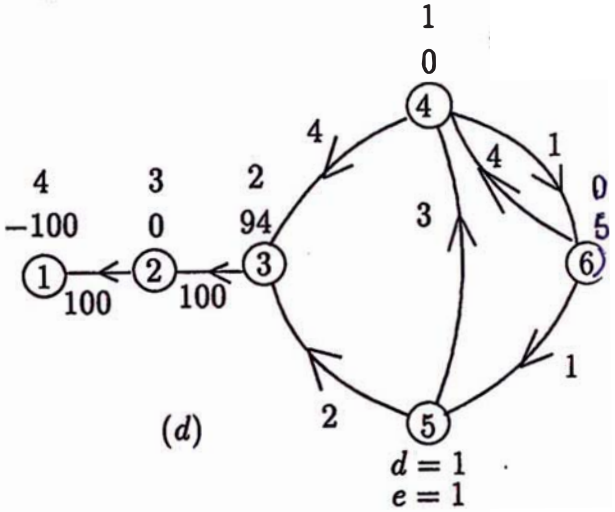
Quinta iteración: A continuación elegimos el nodo activo $i = 5$ y el arco admisible $(5,6)$, sobre el cual enviamos $\delta = 1$ unidad de flujo, obteniendo la red de la figura 3.11(d) .

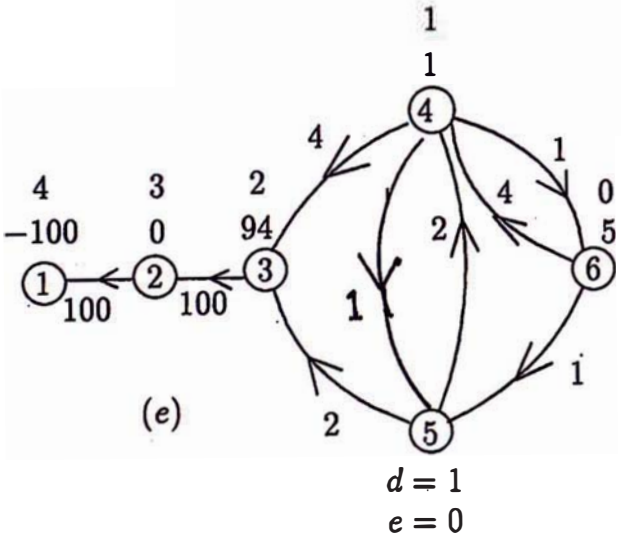

d(i)
e(i)
r_ij
d(j)
e(j)
i
j

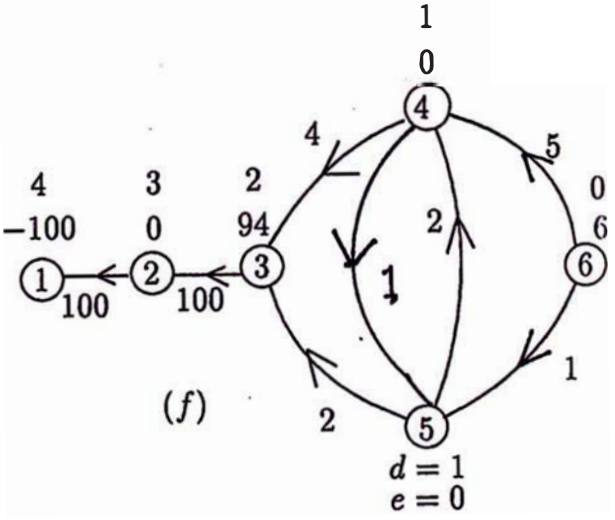

(a)
d = 1
e = 0


(b)
d = 1
e = 2


(c)
d = 1
e = 2


(d)
d = 1
e = 1


(e)
d = 1
e = 0


(f)
d = 1
e = 0

Sexta iteración: Elegimos el nodo activo $i = 5$ nuevamente y el arco admisible $(5,4)$ a través del cual enviamos $\delta = 1$ unidad de flujo, obteniendo la red de la figura 3.11(e).

Séptima iteración: elegimos el nodo activo $i = 4$, arco admisible $(4,6)$ y enviamos a través de dicho arco $\delta = 1$ unidad de flujo, obteniendo la red de la figura 3.11(f).

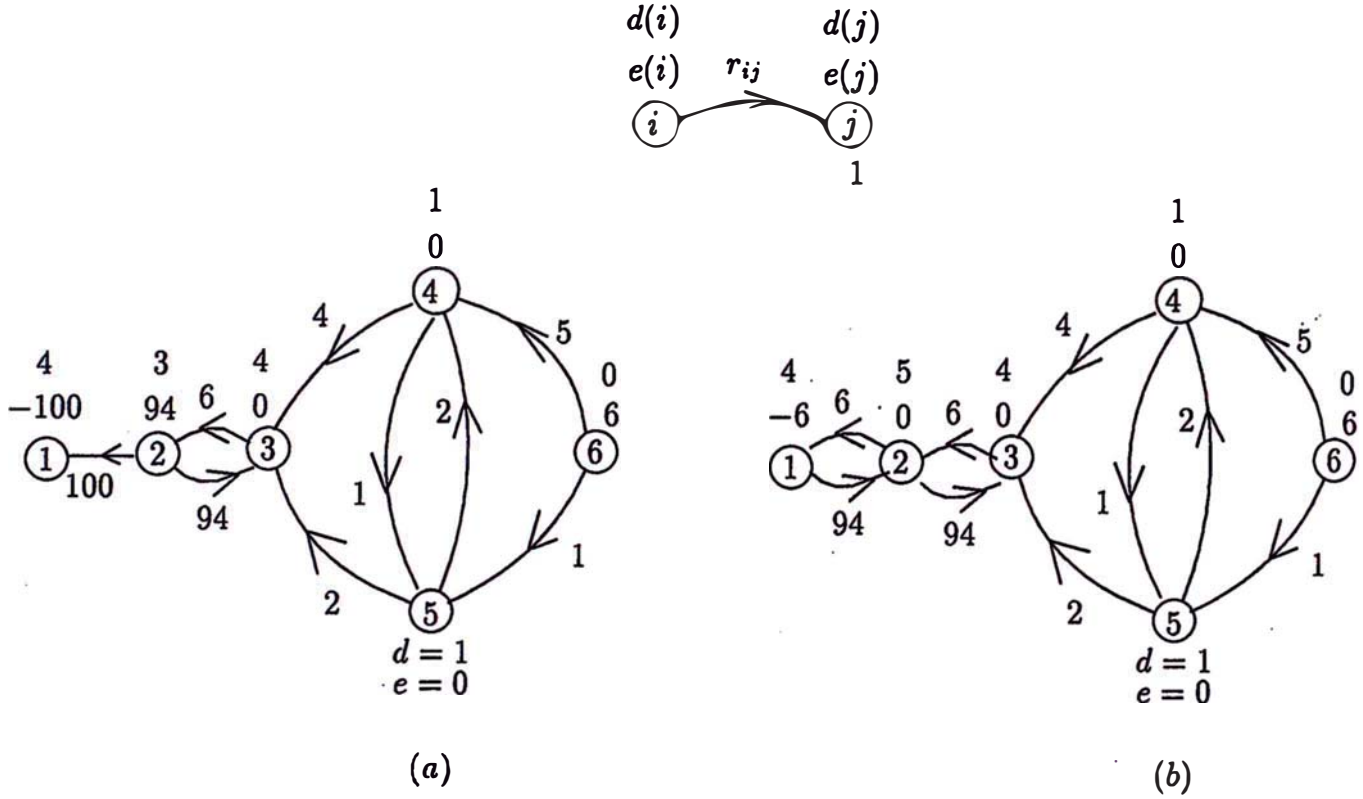


Figura 3.12:

Octava iteración: Elegimos el único nodo activo $i = 3$ (mostrado en la figura 3.11(f)), el cual tiene exceso $e(3) = 94$, dado que este nodo no tiene arco admisible entonces es necesario una operación de cambio de etiqueta, es decir $d(3) = min\{3 + 1\} = 4$, luego $d(3) = 4$, en consecuencia el arco $(3,2)$ es admisible para el nodo activo $i = 3$, entonces se envía $\delta = min\{e(3), r_{32}\} = min\{94, 100\} = 94$ unidades de flujo a través del arco $(3,2)$, obteniendo la red de la figura 3.12(a).

Novena iteración: Elegimos el nodo activo $i = 2$ (figura 3.12(a)), al igual que en la iteración anterior es necesario un cambio de etiqueta para crear un arco admisible, es decir $d(2) = min\{4+1, 4+1\} = 5$, y entonces enviamos $\delta = 94$ unidades de flujo a través del arco admisible $(2,1)$ obteniendo la red de la figura 3.12(b).

Dado que ya no existen nodos activos el algoritmo termina, calculando el flujo máximo según $x_{ij} := max\{u_{ij} - r_{ij}, 0\}$ para todo arco de la red.

Nota: Los nodos origen ($s = 1$) y destino ($t = 6$) tienen excesos -6 y 6 respectivamente, pero en la definición de nodo activo hemos excluido a los nodos origen y destino de ser nodos activos. Además el valor 6 representa el valor del flujo máximo obtenido por el algoritmo "Preflow push"

El algoritmo descrito anteriormente, se denomina **algoritmo genérico**, debido a la libertad que se tiene al escoger los nodos activos, diversas formas de realizar la selección de los nodos activos generan diferentes tipos de implementación del algoritmo genérico, estos serán estudiados brevemente en la siguiente sección. La complejidad de tiempo del algoritmo genérico se enuncia en el siguiente teorema.

Teorema 3.4 *El algoritmo genérico "Preflow push", tiene complejidad de tiempo* $T(m,n) = \mathcal{O}(n^2 m)$

Prueba: ver [3]

Según el teorema anterior la complejidad de tiempo, en el peor caso del algoritmo "Preflow push", todavía no es mejor que la del algoritmo "Shortest augmenting path". En la siguiente sección estudiamos diversas implementaciones de este algoritmo genérico, cuyos tiempos de ejecución son mejorados.

3.4 Diversas Implementaciones del Algoritmo "Preflow push"

Como notamos en la sección anterior, en el algoritmo genérico se tiene la libertad de escoger los nodos activos. Especificando diferentes maneras de seleccionar los nodos activos,

podemos construir diferentes algoritmos con diferentes tiempos de ejecución.

Existen tres tipos específicos de implementación a saber:

Algoritmo "FIFO-preflow-push" Esta implementación, utiliza la estructura de datos llamada **cola** (**FIFO** "first in first out") para almacenar los nodos activos. Entonces los nodos activos forman una cola donde el primer nodo activo que ingresa a la cola es el primero en salir de ella. Seleccionando de esta manera se obtiene un algoritmo cuya complejidad de tiempo es $T(n) = \mathcal{O}(n^3)$

Algoritmo Highest-Label-Preflow-push Este tipo de implementación, escoge el nodo activo i tal que su etiqueta distancia $d(i)$ sea la más alta entre todos los nodos activos. Este algoritmo tiene complejidad de tiempo $T(m,n) = \mathcal{O}(n^2 m^{1/2})$

Algoritmo Excess-scaling-Preflow-push Este algoritmo escoge los arcos admisibles del tipo (i,j), donde $e(i)$ es suficientemente grande y $e(j)$ es suficientemente pequeño. Este tipo de implementación genera un algoritmo cuya complejidad de tiempo es $T(m,n,U) = \mathcal{O}(nm + n^2 Log(U))$

Capítulo 4

Aplicaciones, Programas y Resultados Numéricos

En este capítulo presentamos algunas situaciones reales que pueden ser modeladas como un problema de flujo máximo; la parte de modelaje no es presentada debido a que sería motivo de otro tema. Por otro lado se presenta un programa implementado en el lenguaje Turbo Pascal e inmediatamente después algunas ejecuciones del programa con sus respectivos resultados numéricos.

4.1 Aplicaciones

En esta sección, mencionamos algunas aplicaciones prácticas cuya solución implica plantear y resover un problema del flujo máximo. Los detalles del planteo no se dan por razones de espacio. Además de las aplicaciones que mencionamos existen muchas otras las cuales se pueden encontrar en [3], [5].

4.1.1 Problema de la explotación minera

En operaciones de minería, un problema de gran importancia es determinar el contorno óptimo de la región a explotar. Para ello dividimos la región potencial en bloques, a cada uno de ellos le asociamos un índice i, la tecnología de la que se dispone y la geografía de la

mina, imponen retricciones sobre cómo remover los bloques. Para la optimización a cada bloque i se le asocia el número w_i que representa la ganancia neta obtenida al remover el bloque i (valor del mineral contenido en el bloque menos el costo de explotar y procesar el bloque). En el problema de la explotación minera deseamos identificar el conjunto de bloques que maximiza la ganancia neta total.

4.1.2 Seleccionando terminales de carga

Una compañía de transporte, desea instalar cierto número de terminales de carga, para ello debe escoger de un conjunto S de posibles lugares para instalar los terminales, la compañia tiene el potencial para atraer parte del mercado entre algunos de los pares de terminales (la cual es una cantidad de demanda). Para satisfacer la demanda entre los lugares i y j, la compañia debe instalar terminales en ambos lugares. Supongamos que c_j es el costo de instalar un terminal el el lugar j y p_{ij} es la ganancia neta que se obtiene al satisfacer la demanda entre los lugares i y j. Entonces la compañía de transporte desea determinar donde debe instalar los terminales de carga con el fin de maximizar su ganancia neta (ingreso obtenido al satisfacer las demandas menos el costo de instalar los terminales).

4.1.3 Problema del flujo admisible

Supongamos que cierta mercadería esta disponible en algunos puertos y hay otros puertos que demandan o necesitan de dicha mercadería. Se conoce el stock de mercadería disponible en los puertos, la cantidad requerida por los otros puertos y la máxima cantidad de mercadería que puede ser enviado a través de barcos sobre una ruta particular. El problema es ¿cómo satisfacer todas las demandas usando la mercadería disponible?.

4.1.4 Problema del redondeo de matrices

Sea una matríz $D = [d_{ij}]$ de orden $p \times q$ $(i = 1, 2 \cdots, p \quad j = 1, 2, \cdots, q)$, cuyos elementos son números reales, sean además $\alpha_i = \sum_{j=1}^{q} d_{ij} \quad \forall\ i = 1, 2 \cdots p$ (suma de filas) y $\beta_j =$

$\sum_{i=1}^{p} d_{ij} \quad \forall\, j = 1, 2 \cdots q$ (suma de columnas). El número real a puede redondearse ya sea mediante $\lceil a \rceil$ o mediante $\lfloor a \rfloor$. La elección del tipo de redondeo es nuestra responsabilidad y no afecta a los resultados, por ejemplo elijamos $\lceil a \rceil$ para redondear el número real a.

Se dice que un redondeo es **consistente** cuando:

$$\lceil \alpha_i \rceil = \sum_{j=1}^{q} \lceil d_{ij} \rceil \qquad \forall\, i = 1, 2 \cdots p$$

y

$$\lceil \beta_j \rceil = \sum_{i=1}^{p} \lceil d_{ij} \rceil \qquad \forall\, j = 1, 2 \cdots q$$

El problema del redondeo de matrices consiste en redondear los elementos de la matríz y hallar un redondeo **consistente** si es que existe. Este problema generalmente tiene su aplicación en los censos de población.

4.2 Programa del camino aumentante mas corto

Por razones de espacio tan solo presentamos la implementación de uno de los algoritmos estudiados en el presente trabajo; dicha implementación se realizó utilizando el lenguaje de programación Turbo Pascal versión 7.0, por otro lado en la siguiente sección también se presentan los resultados de la ejecución del programa.

```
(*************************************************************************)
(****                                                                 ****)
(****                      Implementacion del                         ****)
(****                  Algoritmo del Flujo Maximo                     ****)
(****         mediante el metodo del camino amentante mas corto       ****)
(****                                                                 ****)
(*************************************************************************)
Program Shorting_Augmenting_Paths;
uses crt;
const
   MaxNos=50;
```

```
type
   Lista=^nodo;
   nodo= record
           adj:integer;
           prox:Lista
         end;
   Grafo=array[1..MaxNos] of Lista;
   vetor=array[1..MaxNos] of Integer;
   cad40=string[40];
   Matriz=array[1..MaxNos,1..MaxNos] of integer;
var
   U,X:Matriz;
   pred,d:vetor;
   G:Grafo; actual:Lista;
   esp,i,j,source,dest,n:integer;
   nome:cad40;
   arqres:text;
(*************************************************************************)
Procedure Inserta(info:integer; var List,Ult:Lista);
{Inserta al final de la Lista despues de Ult y retorna el valor de Ult }
var
  novo:Lista;
begin
  new(novo);
  novo^.adj:=info;
  novo^.prox:=nil;
  if (List=nil)
    then  List:=novo
    else  Ult^.prox:=novo;
  Ult:=novo
```

```
end;
(*************************************************************************)
Procedure Lectura(var G:grafo; var n:integer; flag:byte; var Nome:cad40);
{ Lee el Grafo a partir del archivo arq cuyo de numbre Nome y lo almacena  }
{ como Lista de Adyacencia si flag=1 entonces lee tambien las capacidades  }
{ sino lee solo la lista de adjacencia, la entrada del grafo tiene un      }
{ formato definido la lectura asume tal formato                            }
var
  arq:text;
  i,vertice,capacidade:integer;
  Ulti:Lista;
begin
  assign(arq,Nome);
  reset(arq);
  for i:=1 to MaxNos do G[i]:=nil;
  i:=1;
  readln(arq,n);
  while not Eof(arq) do
     begin
       while not Eoln(arq) do
          begin
            read(arq,vertice);
            if (vertice=0)
               then
                  begin
                    G[i]:=nil;
                    readln(arq);
                    i:=i+1
                  end
               else
```

```
                    begin
                      if (flag=1)
                            then
                              begin
                                read(arq,capacidade);
                                U[i,vertice]:=capacidade
                              end;
                      Inserta(vertice,G[i],Ulti)
                    end
            end;
          readln(arq);
          i:=i+1
        end;
    close(arq)
end;
(*************************************************************************)
Procedure Remove(info:integer; var List:Lista);
begin
  List:=List^.prox
end;
(*************************************************************************)
Procedure Busca(G:Grafo; origem:integer; var d:vetor);
{ Recibe el grafo G (como Lista de adjacencia Normal o reversa) asi como  }
{ tambien un nodo origem y devolve el vetor 'd' de las distancias exactas }
var
  S,UltS,List,UltList:Lista;
  B:Grafo;
  perte:array[1..Maxnos] of boolean;
  i,j:integer;
begin
```

```
for i:=1 to n do
    begin
       perte[i]:=false;
       d[i]:=n
    end;
List:=nil;S:=nil;
Inserta(origem,List,UltList);
Inserta(origem,S,UltS);
perte[origem]:=true;
pred[origem]:=0;
d[origem]:=0;
B:=G;
while (List<>nil) do
    begin
      i:=List^.adj;
      if (B[i]=nil)
           then Remove(i,List)
           else
             begin
               j:=B[i]^.adj;
               if (not perte[j])
                    then
                       begin
                         d[j]:=d[i]+1; {pred[j]:=i;}
                         Inserta(j,S,UltS);
                         perte[j]:=true;
                         Inserta(j,List,UltList)
                       end;
               Remove(j,B[i])
             end
```

```
      end;
end;
(*************************************************************************)
Procedure CalculaFlujo(U,R:Matriz; var X:Matriz);
{recibe las matrices U:cap; R:cap residual y devuelve el flujo X          }
var i,j:integer;
begin
  for i:=1 to n do
    for j:=1 to n do
        if (U[i,j]-R[i,j] >= 0)
             then
               begin
                 X[i,j]:=U[i,j]-R[i,j];
                 X[j,i]:=0
               end
             else
               begin
                 X[i,j]:=0;
                 X[j,i]:=R[i,j]-U[i,j]
               end
end;
(*************************************************************************)
Procedure EscribeFlujo(X:matriz);
var i,j:integer;
begin
  for i:=1 to n do
     for j:=1 to n do
        if X[i,j]>0 then
             writeln('X[',i,',',j,']=',X[i,j]);
end;
```

```
(*************************************************************************)
Procedure FlujoMax(var G:Grafo; origem,destino:integer; var X:Matriz);
{ recibe un grafo G (como lista de adyacencia) asi como dos nodos  }
{ origen  y destino y devuelve el flujo maximo X                   }
var
  B:Grafo;
  R:Matriz;
  c,i,j:integer;
  delta:integer;
(*-----------------------------------------------------------------------*)
Procedure ActualizarCamino(pred:vetor; var R:Matriz);
{ recibe el vector Pred y actualiza las capacidades residuales del camino }
{ descrito por pred ademas devuelve la matriz R (capacidades residuales)  }
var
   {delta,}atual:integer;
begin
  { calculo de delta=min{R(ij) }
  atual:=pred[destino];
  delta:=R[atual,destino];
  while (atual<>origem) do
      begin
        if (R[pred[atual],atual] < delta)
             then delta:=R[pred[atual],atual];
        atual:=pred[atual]
      end;
  {actualizacion de la red Residual}
  atual:=destino;
  while (pred[atual]<>0) do
      begin
        R[pred[atual],atual]:=R[pred[atual],atual]-delta;
```

```
        R[atual,pred[atual]]:=R[atual,pred[atual]]+delta;
        atual:=pred[atual]
      end
end;
(*-------------------------------------------------------------------*)
function minimo(G:grafo; c:integer):integer;
{ recibe la lista G[c] y calcula   min{ d[j]+1 / (c,j)  G[c]  y  R[c,j]>0 }
var
   aux:Lista;
   k,min:integer;
begin
  aux:=G[c];
  min:=n;
  while aux<>nil do
     begin
       k:=aux^.adj;
       if (R[c,k]>0)
           then
             begin
               if (d[k]+1)<min
                    then min:=d[k]+1
             end;
       aux:=aux^.prox
     end;
  minimo:=min
end;
(*--------------------Bloque Principal de FlujoMax----------------------*)
begin
  {Lee el grafo como lista de adyacencia reversa}
  nome:='c:\bp\bin\combin~1\datos\grafRev7.pas'
```

```
Lectura(G,n,0,nome);
Busca(G,dest,d);{calculo de la distancia exacta d}

{Lee el grafo como lista de adyacencia normal}
nome:='c:\bp\bin\combin~1\datos\grafNor7.pas';
Lectura(G,n,1,nome);
B:=G;
for i:=1 to n  do
   for j:=1 to n do
      R[i,j]:=U[i,j];
c:=origem;
pred[origem]:=0;

while (d[origem] < n) do
     begin
       if (B[c]<>nil)
            then
              begin
                j:=B[c]^.adj;
                remove(j,B[c]);
                if (R[c,j]>0) and (d[c]=d[j]+1) { si (c,j) es admisible}
                       then
                         begin
                           pred[j]:=c;
                           c:=j
                         end;
                if (c=destino)
                       then
                         begin
                           ActualizarCamino(pred,R);
```

```
                        c:=origem;
                      end;
              end
            else {B[c] es vacio}
              begin
                d[c]:=minimo(G,c);
                B[c]:=G[c];
                if (c <> origem)
                     then c:=pred[c]
              end;
      end;
  { calculo del flujo a partir de las capacidades residuales }
  CalculaFlujo(U,R,X)
end;
Procedure escribe_Resultados;
begin
    assign(arqres,'c:\bp\bin\combin~1\result.pas');
  append(arqres);
  writeln(arqres,'Grafo: ', nome);
  writeln(arqres,'Grafo con ',n,' vertices ','{ 1..',n,' }');
  writeln(arqres,'Nodo Origem: ',source);
  writeln(arqres,'Nodo Destino: ',dest);
  writeln(arqres);writeln(arqres);
  for i:=1 to n do
    begin
      actual:=G[i];
      esp:=7;
      write(arqres,i:3,'----> ');
      while actual<>nil do
        begin
```

```
            write(arqres,actual^.adj:3,U[i,actual^.adj]:esp,X[i,actual^.adj]:esp,'
            actual:=actual^.prox;
          end;
        writeln(arqres);
      end;
    writeln(arqres,'-----------------------------------------------------');
    close(arqres)
    {writeln('Flujo Maximo:');
    EscribeFlujo(X);}


end
(*************************** Programa Principal ************************)
begin
  clrscr;
  write(' Ingrese nodo Origem:');
  readln(source);
  write(' Ingrese nodo Destino:');
  readln(dest);
  FlujoMax(G,source,dest,X);
  Escribe_Resultados;
end.
```

4.3 Resultados Numéricos

En esta sección mostramos algunos resultados numéricos, ejecutando el programa anterior para diversos grafos. Cada uno de los grafos es presentado en su forma de lista de adyacencia (implementación más práctica). En cada lista de adyacencia se tiene el vértice adyacente, la capacidad y el flujo; tal como se explica para los dos primeros resultados, para el resto el significado es el mismo.

```
Grafo: c:\bp\bin\combin~1\datos\grafNor.pas
Grafo con 4 vertices { 1..4 }
Nodo Origem: 1
Nodo Destino: 4


  1 ----->   2  10000  10000   -->   3  10000  10000   -->.
  2 ----->   4  10000  10000   -->.
  3 ----->   2      1      0   -->   4  10000  10000   -->.
  4 ----->.
```

La primera fila tiene el siguiente significado: El arco (1,2) tiene capacidad $u(1,2) = 10000$ unidades y flujo $x(1,2) = 10000$ unidades; similarmente el arco (1,3). Por otro lado en la tercera fila se observa que el arco (3,2) tiene capacidad $u(3,2) = 1$ y por él atravieza un flujo de $x(3,2) = 0$ y asi para todas las filas.

```
Grafo: c:\bp\bin\combin~1\datos\grafNor1.dat
Grafo con 12 vertices { 1..12 }
Nodo Origen: 1
Nodo Destino: 12


 1----->  2  2  1 -->  3  5  3 -->  4  7  6 -->  5  1  1-->  6  3  3 -->.
 2----->  7  1  1 -->  3  2  0 -->.
 3----->  7  3  2 -->  8  1  1 -->  4  3  0 -->.
 4----->  8  3  3 -->  9  1  1 -->  5  4  2 -->.
 5----->  9  2  2 --> 10  1  1 -->  6  6  0 -->.
 6----> 10  4  2 --> 11  1  1 -->.
 7----> 12  3  3 -->  8  1  0 -->.
```

```
 8----> 12  2  2 -->  9  2  2 -->.
 9----> 12  7  5 --> 10  3  0 -->.
10----> 12  1  1 --> 11  4  2 -->.
11----> 12  3  3 -->.
12---->.
```

Interpretemos la primera fila del resultado anterior (grafNor1.dat):
el arco (1,2) tiene capacidad $u(1,2) = 2$ y el flujo es $x(1,2) = 1$
el arco (1,3) tiene capacidad $u(1,3) = 5$ y el flujo es $x(1,3) = 3$
el arco (1,4) tiene capacidad $u(1,4) = 7$ y el flujo es $x(1,4) = 6$
el arco (1,5) tiene capacidad $u(1,5) = 1$ y el flujo es $x(1,5) = 1$
el arco (1,6) tiene capacidad $u(1,6) = 3$ y el flujo es $x(1,6) = 3$
Similar significado tienen todas las filas del resultado.

Además el símbolo $'- -- >'$ significa próximo elemento de la lista de adyacencia y el símbolo $'- -- > .'$ significa fin de lista.

```
Grafo: c:\bp\bin\combin~1\datos\grafNor2.dat
Grafo con 14 vertices { 1..14 }
Nodo Origen: 1
Nodo Destino: 14


  1---->   2  20000      7   -->.
  2---->   3  20000      7   -->.
  3---->   4  20000      7   -->.
  4---->   5  20000      7   -->.
  5---->   6  20000      7   -->.
  6---->   7      1      1   -->   8      1      1   -->   9      1      1
     -->  10      1      1   -->  11      1      1   -->  12      1      1
```

```
     --> 13      1      1   -->.

  7----> 14      1      1   -->.
  8----> 14      1      1   -->.
  9----> 14      1      1   -->.
 10----> 14      1      1   -->.
 11----> 14      1      1   -->.
 12----> 14      1      1   -->.
 13----> 14      1      1   -->.
 14---->.
-------------------------------------------------------
Grafo: c:\bp\bin\combin~1\datos\grafNor3.dat
Grafo con 6 vertices { 1..6 }
Nodo Origen: 1
Nodo Destino: 6


  1----->  2   3   1 -->  4   2   2 -->  3   3   2 -->.
  2----->  5   4   1 -->.
  3----->  4   1   0 -->  6   2   2 -->.
  4----->  2   1   0 -->  6   2   2 -->.
  5----->  4   1   0 -->  6   1   1 -->.
  6---->.
-------------------------------------------------------------------------
Grafo: c:\bp\bin\combin~1\datos\grafNor4.dat
Grafo con 6 vertices { 1..6 }
Nodo Origen: 1
Nodo Destino: 6
```

```
  1----->   2   5   5 -->   3   8   7 -->   4   3   3 -->   5   9   8 -->.
  2----->   6  12  10 -->   3   2   0 -->.
  3----->   6   2   2 -->   2   5   5 -->.
  4----->   6   7   7 -->   5   3   0 -->.
  5----->   4   6   4 -->   6   4   4 -->.
  6----->.
-------------------------------------------------------------------------
Grafo: c:\bp\bin\combin~1\datos\grafNor5.dat
Grafo con 7 vertices { 1..7 }
Nodo Origen: 1
Nodo Destino: 7


  1----->   4  10   8 -->   2  20  14 -->   3  15   6 -->.
  2----->   4   4   0 -->   5   5   5 -->   6   9   9 -->.
  3----->   2   5   0 -->   6   6   6 -->.
  4----->   5   8   8 -->.
  5----->   6  25   3 -->   7  10  10 -->.
  6----->   5   5   0 -->   7  30  18 -->.
  7----->.
-------------------------------------------------------------------------
Grafo: c:\bp\bin\combin~1\datos\grafNor6.dat
Grafo con 9 vertices { 1..9 }
Nodo Origen: 1
Nodo Destino: 9


  1----->   2   5   4 -->   3   6   6 -->   4   8   6 -->   5  10  10 -->.
  2----->   6   4   4 -->   7   3   0 -->.
  3----->   7   7   6 -->.
```

```
 4---->   7   9   6 -->.
 5---->   7   4   4 -->   8  10   6 -->.
 6---->   9   5   4 -->.
 7---->   9  16  16 -->.
 8---->   9   8   6 -->.
 9---->.
```

```
Grafo: c:\bp\bin\combin~1\datos\grafNor7.dat
Grafo con 14 vertices { 1..14 }
Nodo Origen: 1
Nodo Destino: 14

 1---->   2      2      1   -->   3      2      1   -->   4      2      1
    -->   5      2      1   -->   6      2      1   -->   7      2      1
    -->.

 2---->   8  10000      1   -->   9  10000      0   -->  10  10000      0
    -->  11  10000      0   -->  12  10000      0   -->  13  10000      0
    -->.

 3---->   8  10000      0   -->   9  10000      1   -->  10  10000      0
    -->  11  10000      0   -->  12  10000      0   -->  13  10000      0
    -->.

 4---->   8  10000      0   -->   9  10000      0   -->  10  10000      1
    -->  11  10000      0   -->  12  10000      0   -->  13  10000      0
    -->.

 5---->   8  10000      0   -->   9  10000      0   -->  10  10000      0
```

```
    -->  11  10000      1   -->  12  10000      0   -->  13  10000      0
    -->.

 6---->   8  10000      0   -->   9  10000      0   -->  10  10000      0
    -->  11  10000      0   -->  12  10000      1   -->  13  10000      0
    -->.

 7---->   8  10000      0   -->   9  10000      0   -->  10  10000      0
    -->  11  10000      0   -->  12  10000      0   -->  13  10000      1
    -->.

 8---->  14      1      1   -->.
 9---->  14      1      1   -->.
10---->  14      1      1   -->.
11---->  14      1      1   -->.
12---->  14      1      1   -->.
13---->  14      1      1   -->.
14---->.
```

Capítulo 5

Conclusiones

A modo de resumen presentamos una tabla comparativa , con todos los algoritmos estudiados y algunos tan solo presentados.

Empezamos estudiando el algoritmo genérico de etiquetas, estudiado en el capítulo 2, este algoritmo tiene la limitación de realizar en el peor caso nU aumentaciones (demasiado alto) debido a que cada aumentación puede llevar poca cantidad de flujo. En el capítulo 3 estudiamos dos estrategias para mejorar esta limitación; la primera es buscar arcos con capacidad alta ("capacity scaling") y la segunda buscar caminos con el menor número de arcos ("shortest augmenting path").

Una limitación inherente de todos los métodos anteriores es que ellos siempre envían flujo a través de caminos aumentantes y por lo tanto la complejidad de ésta operación será en el peor caso $\mathcal{O}(n)$. Los algoritmos llamados "Preflow push" vistos en el capítulo 3 mejoran esta limitación pues ellos envían flujo a través de arcos en vez de enviar flujo a través de caminos. Esto puede traducirse en términos prácticos remarcando que los algoritmos de caminos aumentantes envían flujo a través de un solo camino a la vez, mientras que los algoritmos "Preflow push" envian flujo a través de varios caminos simultaneamente.

La siguiente tabla muestra un resumen de los diversos algoritmos.

Algoritmo	Complejidad	Características
De etiquetas	$\mathcal{O}(nmU)$	1) facil de implementar 2) complejidad no polinomial, ineficiente
"Capacity scaling"	$\mathcal{O}(nmLog(U))$	1) Caso especial del alg. de etiquetas 2) busca arcos con capacidad grande
"Shortest aug. path"	$\mathcal{O}(n^2m)$	1) Caso especial del alg. de etiquetas 2) busca caminos con pocos arcos 3) usa función distancia válida para identificar caminos aumentantes 4) fácil de implementar, muy eficiente
"Preflow push"	$\mathcal{O}(n^2m)$	1) mantiene preflujo en cada paso 2) utiliza nodos activos 3) muy flexible, examina nodos activos en cualquer orden 4) difícil de implementar
"FIFO-preflow"	$\mathcal{O}(n^3)$	1) caso especial del alg. genérico 2) examina nodos activos en onden FIFO 3) muy flexible, examina nodos activos en cualquer orden 4) difícil de implementar
"Highest-label-preflow"	$\bullet(n^2m^{1/2})$	1) caso especial del alg. genérico 2) examina nodos activos con la etiqueta mas alta 3) algoritmo mas eficiente en la práctica
"Excess-scaling-preflow"	$\mathcal{O}(nm + n^2Log(U))$	

Tabla 5.1: Comparación de los diferentes algoritmos

Bibliografía

[1] A.V. Aho, J.E. Hopcroft, J.D. Hullman, The Design and Analysis of Computer Algorithms, Addison Wesley, Reading Mass. 1974.

[2] A.V. Aho, J.E. Hopcroft, J.D. Hullman, Data Structures and Algorithms, Addison Wesley, Reading Mass. 1983.

[3] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows Theory, Algorithms and Applications. Prentice Hall, Inc New Jersey 1993.

[4] J.A. Bondy, U.S. Murty, Graph Theory with Applications, American Elsevier, New York 1976.

[5] T.H. Cormen, C.E. Leiserson y R.L. Rivest, Introduction to Algorithms. MIT Press and McGraw-Hill, New York 1990.

[6] E.A. Dinic, algorithm for Solution of a Problem of Maximum Flow in Networks with Power estimation. Soviet Mathematics Doklady 11, 1277-1280 1970.

[7] L.R. Ford, D.R. Fulkerson, Flows in Networks, Princeton University Press New York 1962.

[8] L.R. Ford, Network Flow Theory. Report P-923, Rand Corp. Santa Mónica 1956.

[9] L.R. Ford, D.R. Fulkerson, Maximal Flow through a Network. Canadian Journal of Mathematics 8, 399-404 1956.

[10] D.E. Knuth, The Art of Computer Programming 1, Fundamental Algorithms, Addison Wesley, Reading Mass. 1968.

[11] D.E. Knuth, The Art of Computer Programming 3, Sorting and Searching, Addison Wesley, Reading Mass. 1973.

[12] J.L. Szwarcfiter, L. Markenzon, Estruturas de Dados e seus Algoritmos, Editora, Rio de Janeiro 1994.

[13] N. Wirth, Algorithms and Data Structures, Prentice Hall, Englewood Cliffs, New Jersey 1986.