











Flows I: Max Flow

Contenido

1. Flows	
2. Residual networks	
3. Cuts	
4. Ford-Fulkerson algorithm	

Contenido

1. Flows	
2. Residual networks	
3. Cuts	
4. Ford-Fulkerson algorithm	

Motivación

- ❑ Supongamos que tenemos una red de tuberías en donde cada tubería tiene una capacidad máxima de litros por segundo. Dada una fuente desde donde generamos agua y un sumidero a donde queremos que llegue, ¿cuál es el máxima cantidad de agua que podemos mandar por cada segundo?
- ❑ Asumamos que por cada tubería siempre pasa una cantidad no negativa de agua
- ❑ Asumamos que los nodos intermedios (que no son la fuente ni el sumidero) tienen la restricción que toda el agua que entra tiene que salir.

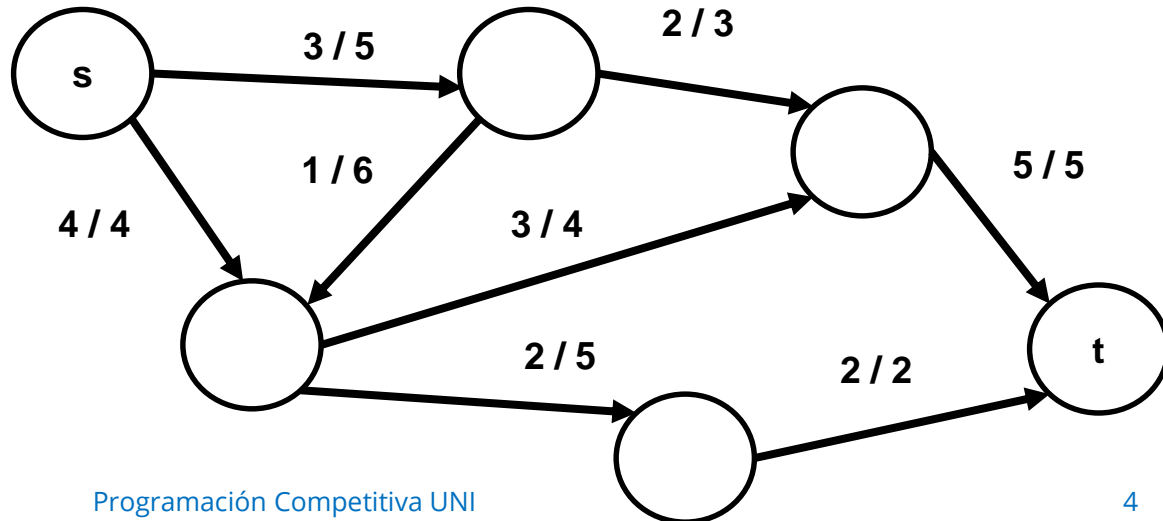


Ejemplo:

Leyenda

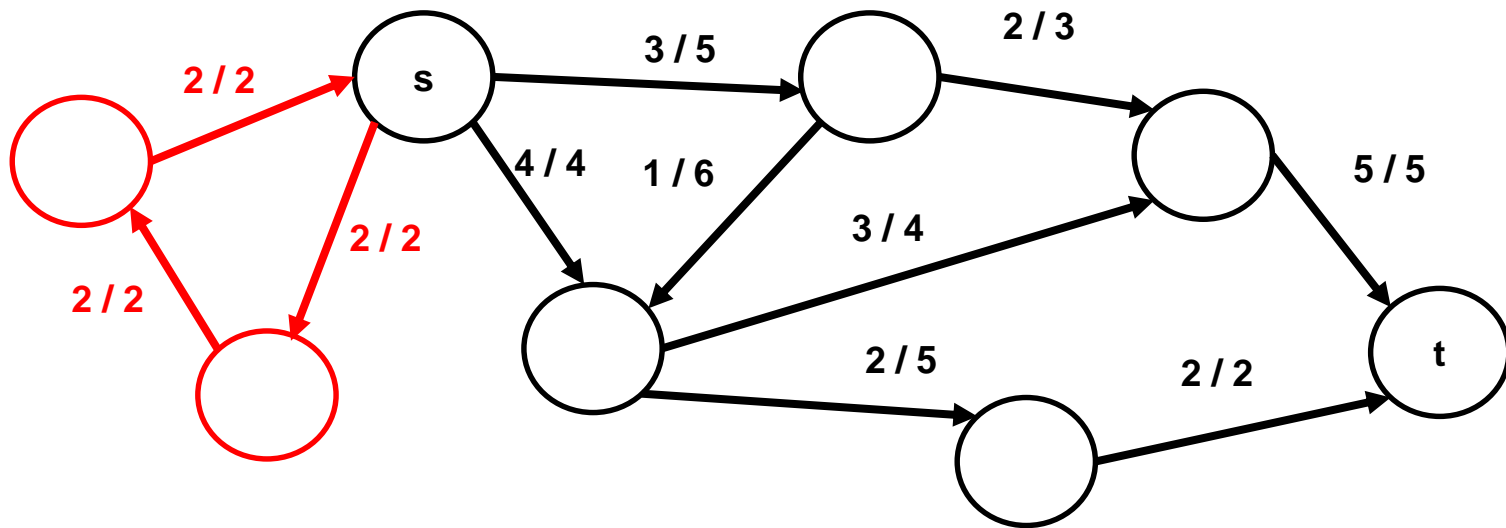
Flujo / capacidad

La red manda 7
de flujo



Motivación

- ❑ Nuestra definición tiene que contemplar el caso esquina donde existe flujo que sale de la fuente pero vuelve a entrar a ella. Este flujo no debería contar en lo que se envía. Por ejemplo, en la siguiente red, solo 7 de flujo llega de s a t cada segundo. Podemos verlo como “flujo neto”.



Conceptos – Flow Network

- ❑ Definimos una red de flujo como un grafo dirigido $G(V, E)$ en donde cada arista $e = (u, v)$ tendrá una capacidad no-negativa, es decir $c(u, v) \geq 0$ para todo $(u, v) \in E$.
- ❑ La red tendrá 2 nodos importantes s y t , la fuente (*source*) y el sumidero (*sink*)
- ❑ Asumiremos, por el momento, que si $(u, v) \in E$, entonces $(v, u) \notin E$ (no antiparallel edges).

Definiremos un $(s - t)$ *flow* como una función $f: E \rightarrow \mathbb{R}$ que satisface las siguientes condiciones.

- ❑ **Capacity constraint:** $\forall (u, v) \in E, 0 \leq f(u, v) \leq c(u, v)$
- ❑ **Flow conservation:** $\forall u \in V - \{s, t\}, \sum_{in} f(in, u) = \sum_{out} f(u, out)$

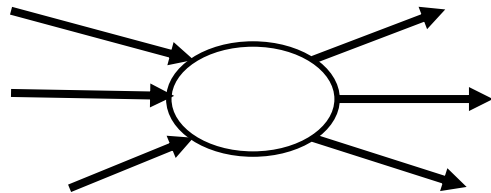
Por simplicidad, podemos extender el dominio de las funciones f, c a todo par de vértices y definir que cuando $(u, v) \notin E, f(u, v) = c(u, v) = 0$. Entonces el flujo con nuevo dominio sería $f: (V \times V) \rightarrow \mathbb{R}$

Definamos también el flujo neto de un nodo: $net(u) = \sum_{out} f(u, out) - \sum_{in} f(in, u)$

Por último, se define el valor de un flujo como el flujo neto en s .

$$|f| = net(s) = \sum_{out} f(s, out) - \sum_{in} f(in, s)$$

El objetivo es maximizar $|f|$ (max flow).



Observaciones

□ **Propiedad 1:** $\forall u \in V - \{s, t\}, \text{net}(u) = 0$ (deducción directa de **flow conservation**)

□ **Propiedad 2:** $|f| = \text{net}(s) = -\text{net}(t)$

Demostración:

Para cada arista $(u, v) \in E$, el valor de su flujo $f(u, v)$ aparece solo en el flujo neto de u como positivo y en el de v como negativo.

$$\Rightarrow \sum_{u \in V} \text{net}(u) = 0$$

$$\Rightarrow \text{net}(s) + \text{net}(t) + \sum_{u \in V - \{s, t\}} \text{net}(u) = 0$$

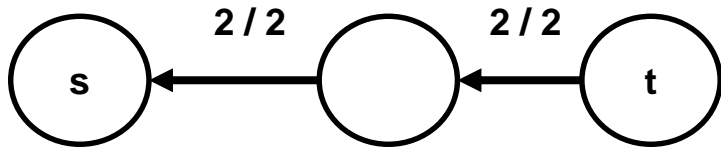
$$\Rightarrow \text{net}(s) + \text{net}(t) = 0$$

$$\Rightarrow \text{net}(s) = -\text{net}(t) \blacksquare$$

Podemos verlo como que el flujo neto que sale de s es igual al flujo neto que entra a t

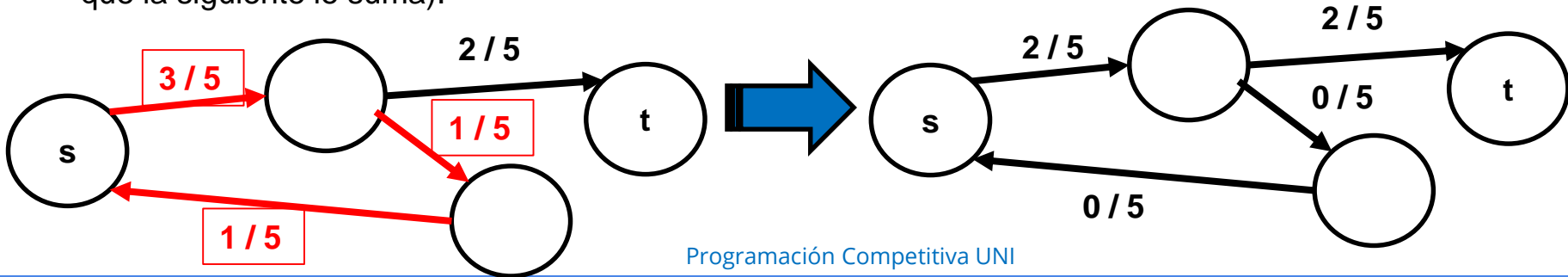
Observaciones

- ❑ El valor de un flujo puede ser negativo. No hay ninguna restricción que lo impida. **Ejemplo:**



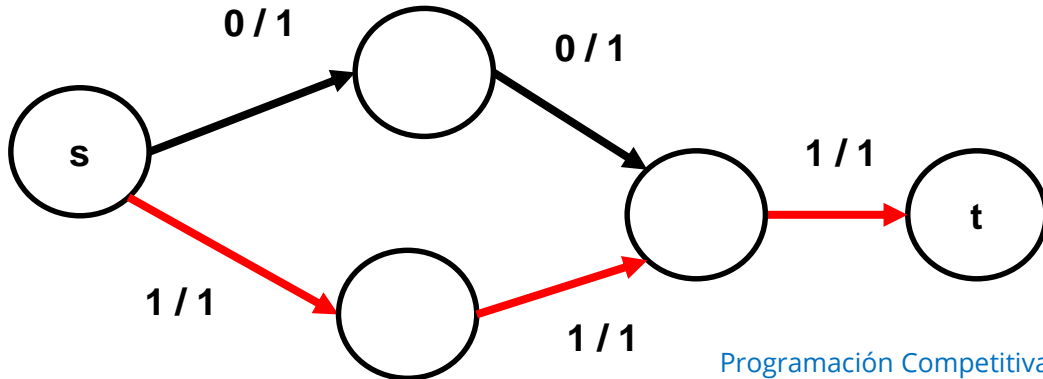
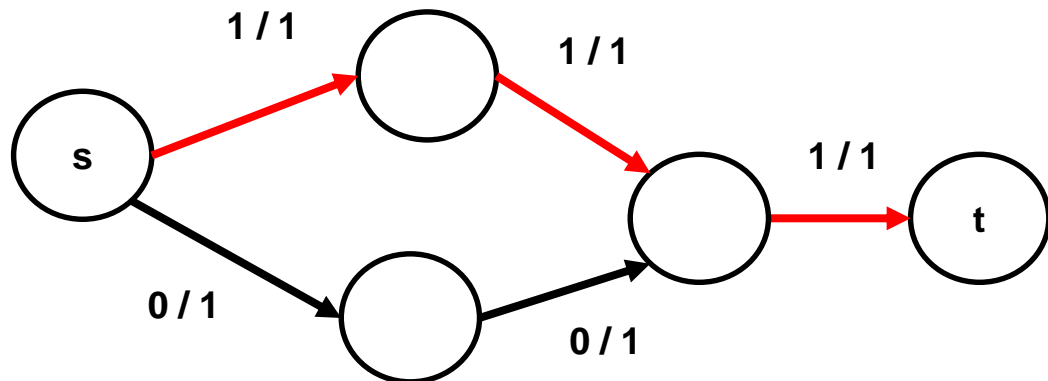
El valor del flujo es $|f| = -2$ pero cumple las 2 constraints (**capacity constraint** y **flow conservation**)

- ❑ Sea f^* la función de flujo óptima, entonces $|f^*| \geq 0$. Debido a que siempre puedo no enviar nada y es mejor que un valor negativo.
- ❑ Los ciclos de flujo son válidos pero no sirven para nada. La razón es que al disminuir el flujo del ciclo hasta eliminarlo, todos los flujos netos se mantendrán igual (porque una arista del ciclo resta mientras que la siguiente le suma).







Observaciones

❑ El max flow no es necesariamente único



Contenido

1. Flows	
2. Residual Networks	
3. Cuts	
4. Ford-Fulkerson algorithm	

Naive algorithm

❑ ¿Cuál es el algoritmo más simple que se nos puede ocurrir?

Algorithm 1: Greedy flow

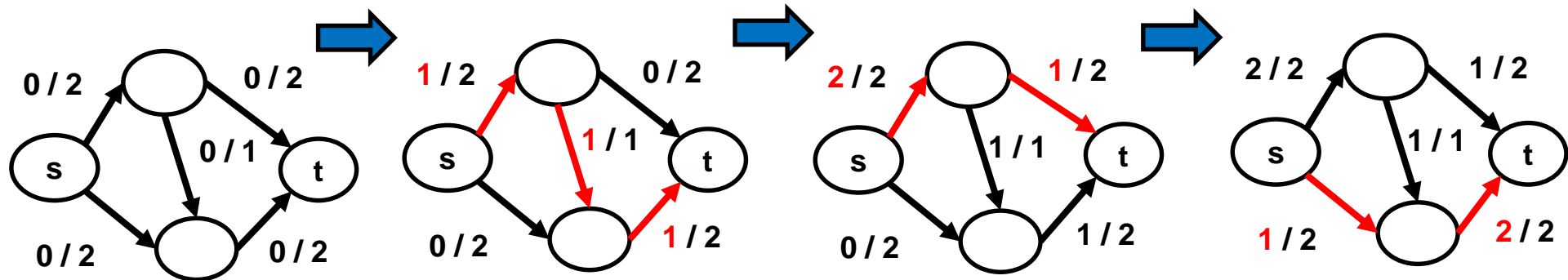
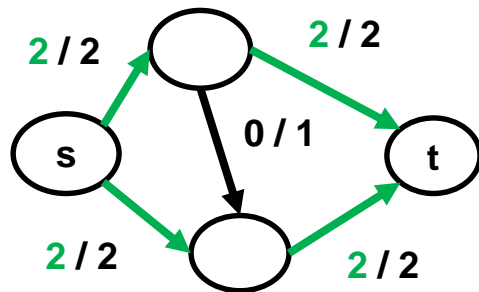
input : $G(V, E, c)$, $source : s$, $sink : t$

output: max flow: f^*

- 1 Initialize flow f^* to 0
 - 2 **while** there exists a path P to send flow from s to t **do**
 - 3 | Send flow through P and update f^*
 - 4 **end**
 - 5 **return** f^*
-

Lamentablemente
es incorrecto

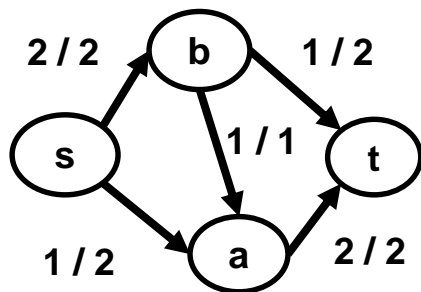
Max flow



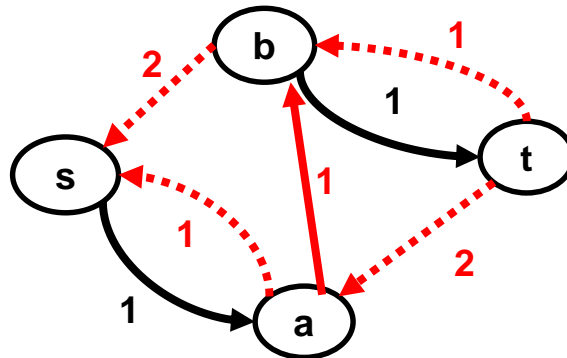
Residual networks - Intuición

- ❑ Necesitamos una forma de deshacer algunas decisiones erradas, es decir “redireccionar” el flujo.
- ❑ Utilizaremos el artificio de crear aristas en la dirección contrario para tratar de revertir el flujo.
- ❑ En cada arista pondremos la cantidad máxima de flujo que todavía podemos mandar (o revertir) a través de ella. Esto se conoce como **residual capacity** y a la red como **residual network**.

Original flow network



Residual network



Nota que en esta nueva red, podemos tomar el camino $s \rightarrow a \rightarrow b \rightarrow t$. Las aristas $s \rightarrow a$ y $b \rightarrow t$ ya existían en el grafo original, pero la arista $a \rightarrow b$, no, y tomarla es equivalente a deshacer nuestra acción anterior y redireccionar ese flujo hacia $b \rightarrow t$. Esto mejora nuestra respuesta en 1. Este tipo de path se llama **augmenting path**.

Residual networks - Definición

❑ **Residual capacity:** Denotada como $c_f(u, v)$

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & (u, v) \in E \\ f(v, u), & (v, u) \in E \\ 0, & \text{en otro caso} \end{cases}$$

Tengamos en cuenta que como estamos asumiendo que no hay anti parallel edges, entonces el par (u, v) siempre cae en alguno de esos 3 casos. Además, como $f(u, v) \leq c(u, v) \Rightarrow c_f(u, v) \geq 0$

❑ **Residual networks:** La red residual de G con el flujo f , denotado como $G_f = (V, E_f)$, es el grafo que consiste de los mismo vértices que G y sus aristas consisten en todos los pares de nodos (u, v) que tengan capacidad residual positiva. G_f contiene tanto aristas originales de G como algunas aristas revertidas.

❑ **Augmenting path:** Es un path (simple) desde s a t en la red residual G_f . También se define la capacidad de un augmenting path P como la mínima capacidad residual de sus aristas, que representa lo máximo de flujo que se puede mandar por este path. Es decir

$$c_f(P) = \min_{(u,v) \in P} c_f(u, v)$$

$$c_f(P) > 0$$

Augmentation

- **Definición:** Sea f una función de flujo y sea P un augmenting path en G_f . Definiremos la operación augmentation $f + P$ como el resultado de actualizar f enviando flujo a través de P .

$$(f + P)(u, v) = \begin{cases} f(u, v) + c_f(P), & (u, v) \in P \wedge (u, v) \in E \\ f(u, v) - c_f(P), & (v, u) \in P \wedge (u, v) \in E \\ f(u, v), & (u, v) \notin P \wedge (v, u) \notin P \end{cases}$$

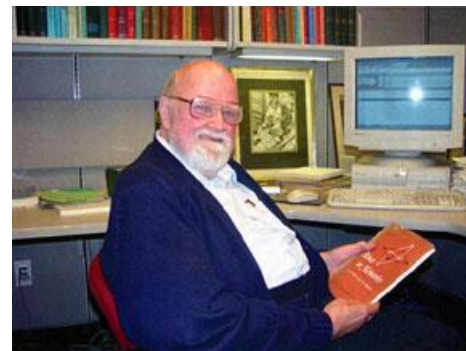
- Nota que estos son los 3 únicos casos, debido a que si la arista $(u, v) \in P$, se tiene que cumplir que $(u, v) \in E$ o $(v, u) \in E$, de lo contrario tendríamos $c_f(u, v) = 0$ y no sería parte de la red residual (contradicción).
- Basado en las definiciones anteriores, podemos estar tentados a hacer un algoritmo que busque *augmenting paths* para realizar *augmentation* hasta que ya no encuentre más. Sin embargo, debemos demostrar que cada vez que hacemos un *augmentation* al flujo actual, obtenemos una función de flujo válida y que el flujo resultante es óptimo.

Ford-Fulkerson algorithm

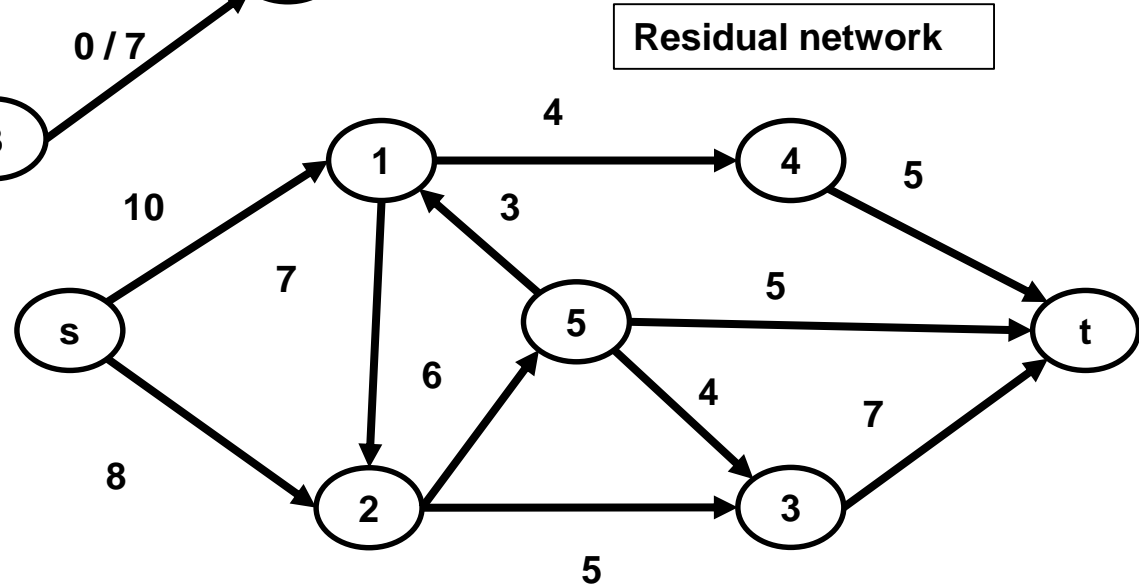
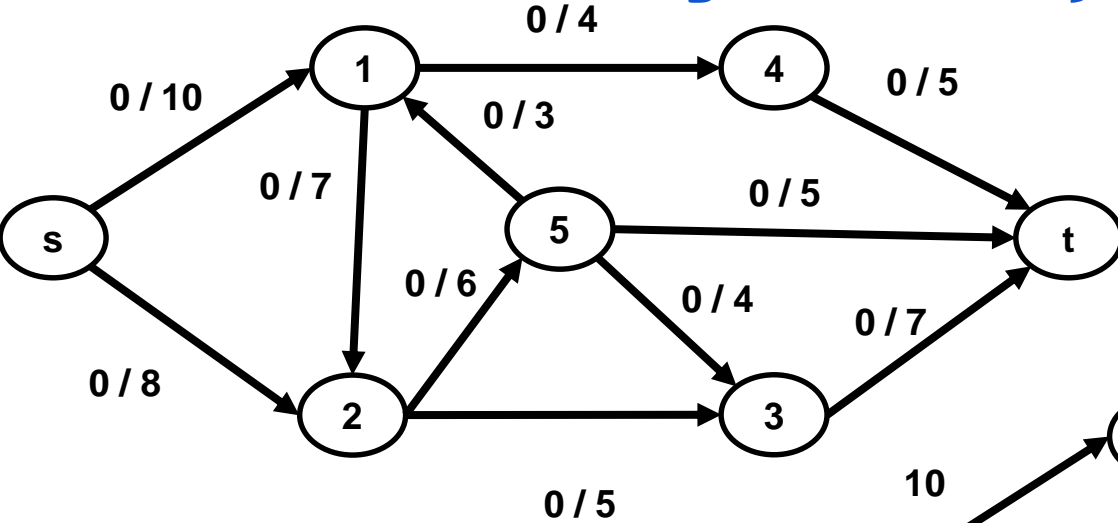
- En 1955, Lester Ford (que también contribuyó en algoritmos de Shortest Path) y Delbert Fulkerson publicaron “*A Simple Algorithm for Finding Maximal Network Flows and an Application to the Hitchcock Problem*”, en donde plasmaron su algoritmo greedy.

Algorithm 2: Ford-Fulkerson algorithm

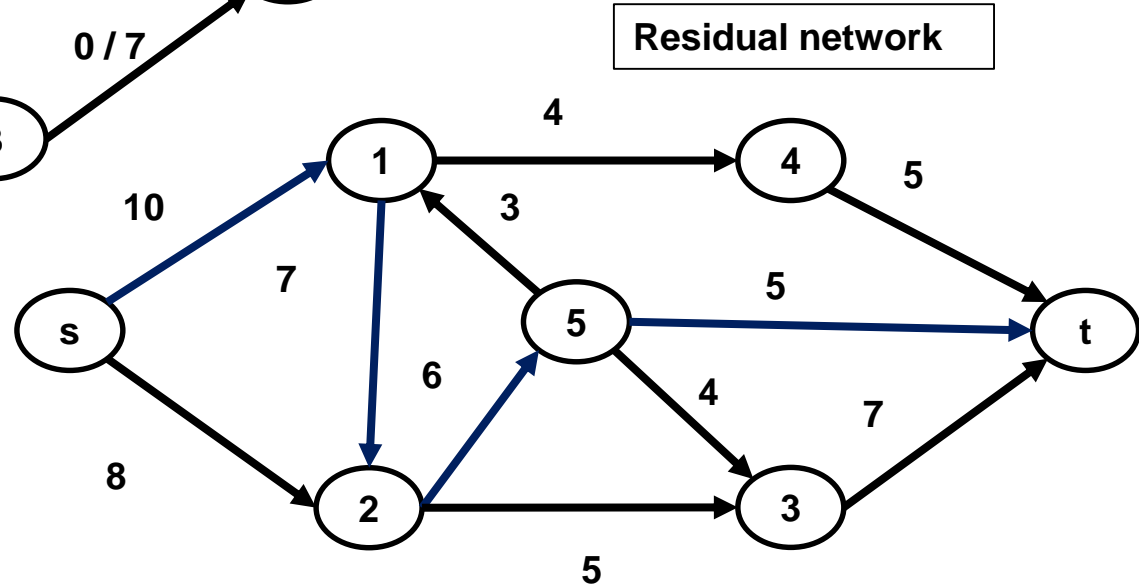
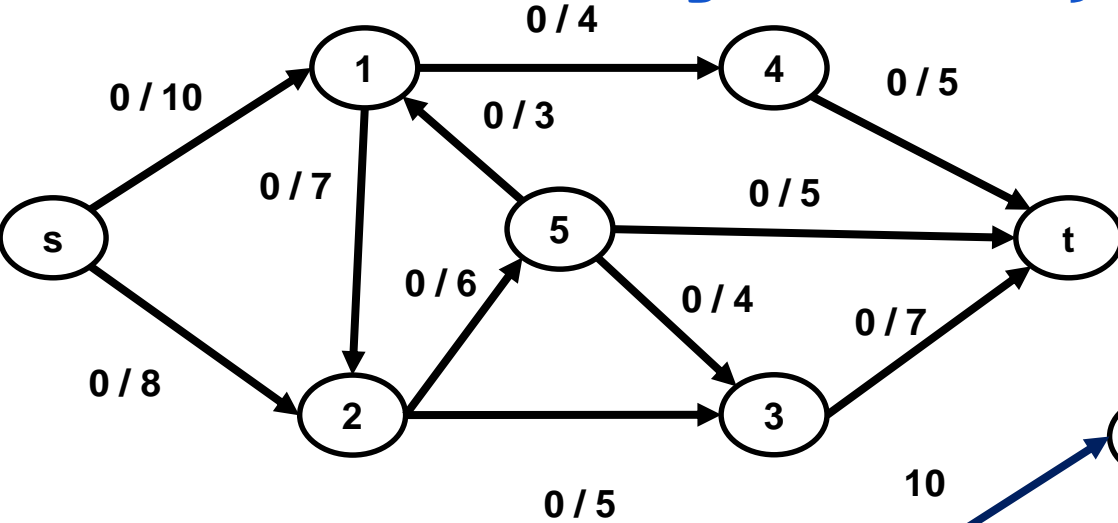
```
input :  $G(V, E, c)$  , source :  $s$ , sink :  $t$ 
output: max flow:  $f^*$ 
1 Initialize flow  $f^*$  to 0
2 while there exists a path  $P$  in the residual network  $G_f$  from  $s$  to  $t$  do
3    $c_f(P) = \min_{(u,v) \in P} c_f(u, v)$ 
4   foreach edge  $(u, v) \in P$  do
5     if  $(u, v) \in E$  then
6        $f(u, v) = f(u, v) + c_f(P)$ 
7     else
8        $f(v, u) = f(v, u) - c_f(P)$ 
9     end
10  end
11 end
12 return  $f^*$ 
```



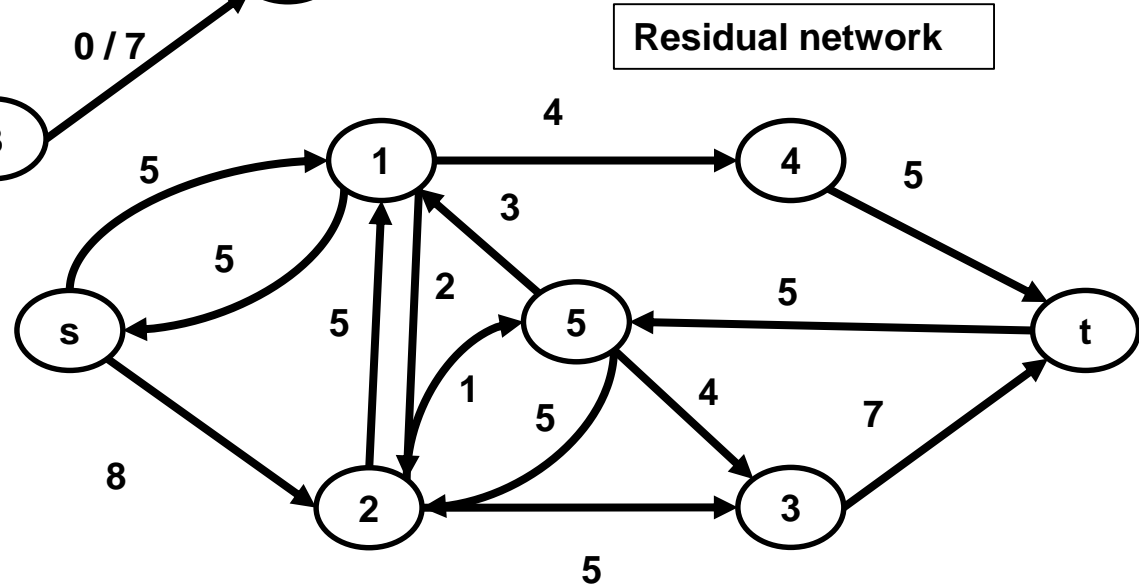
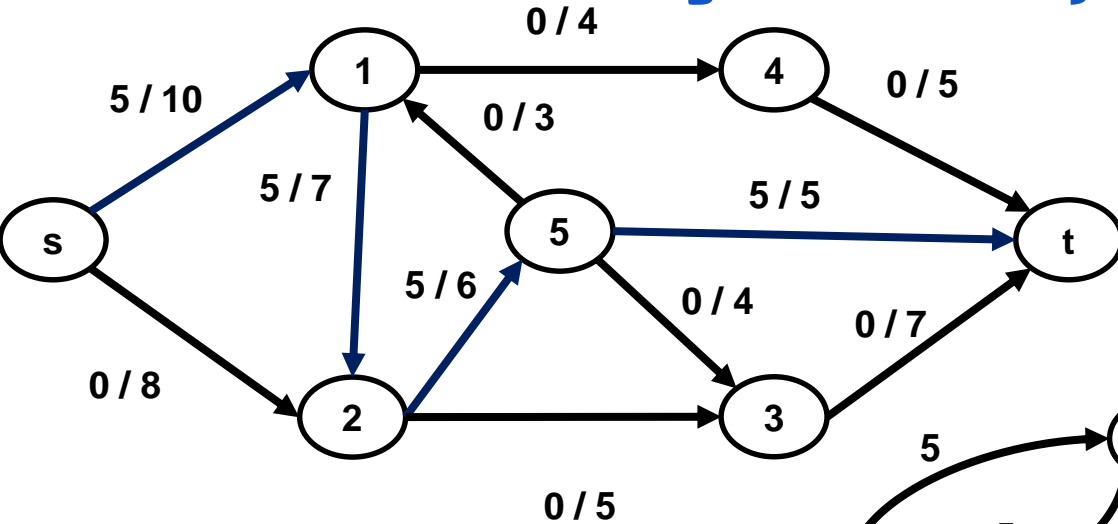
Ford-Fulkerson algorithm - Ejemplo



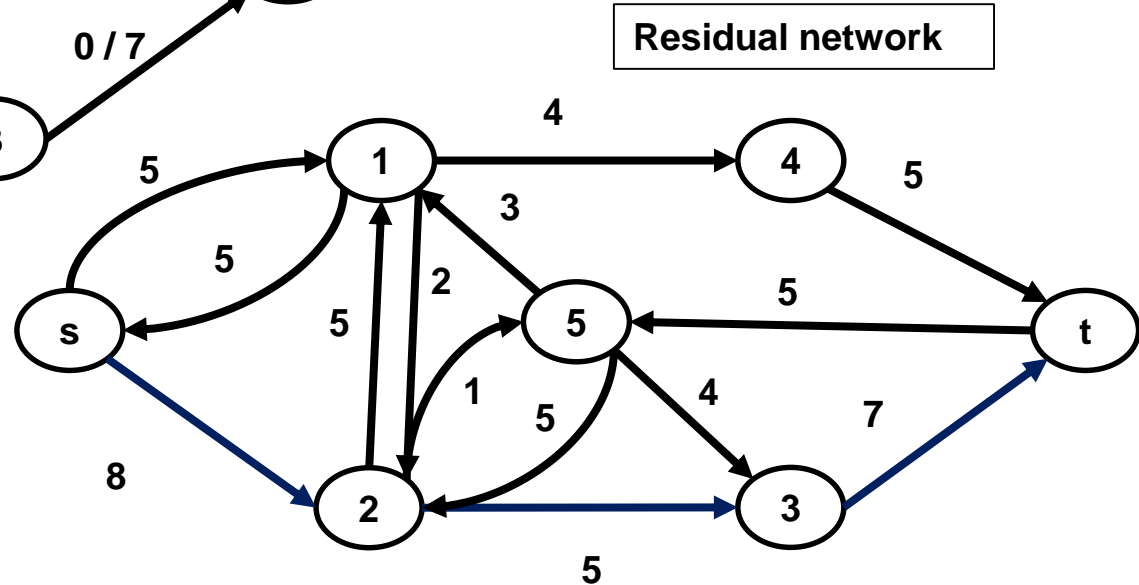
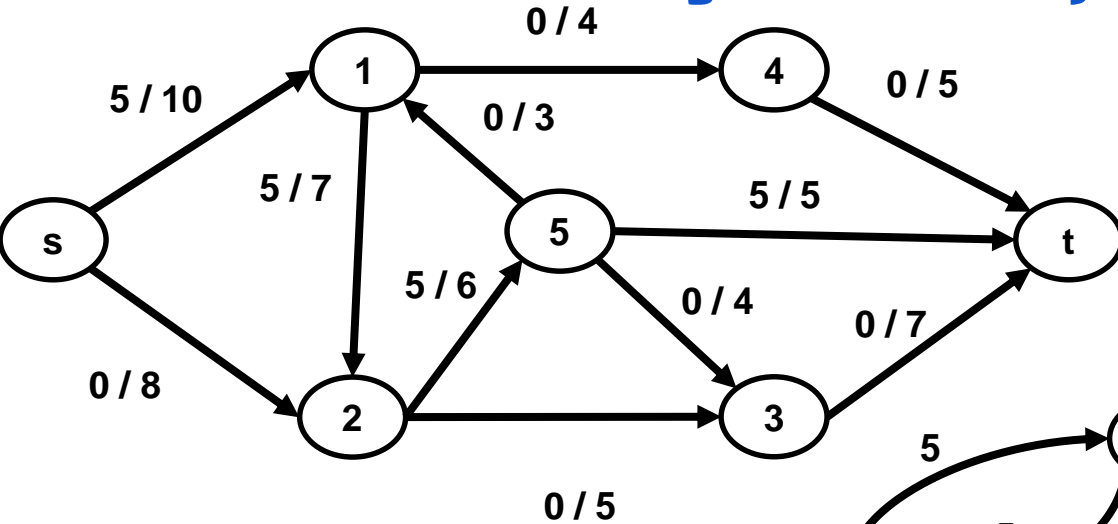
Ford-Fulkerson algorithm - Ejemplo



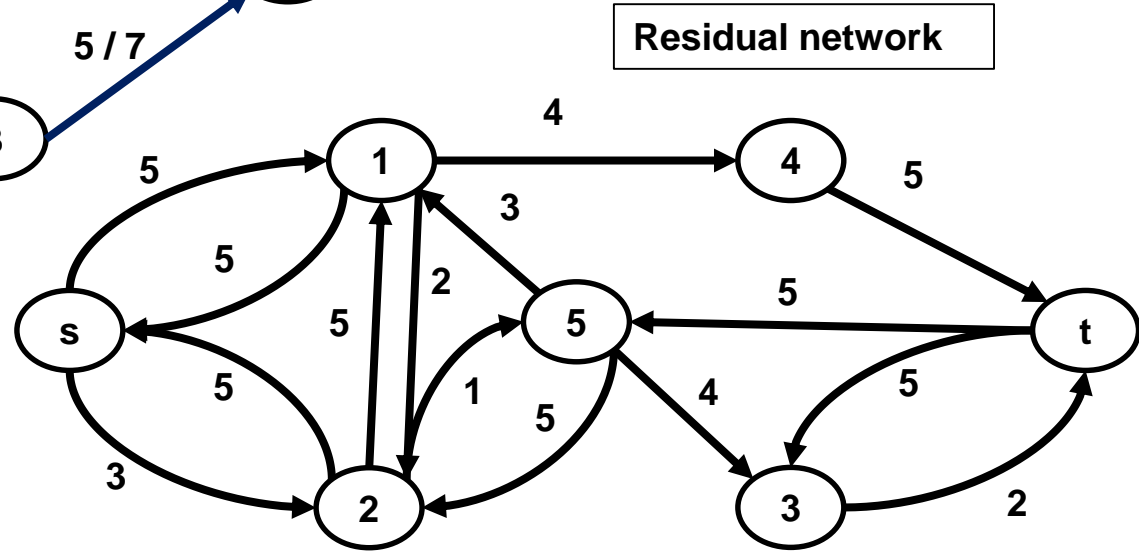
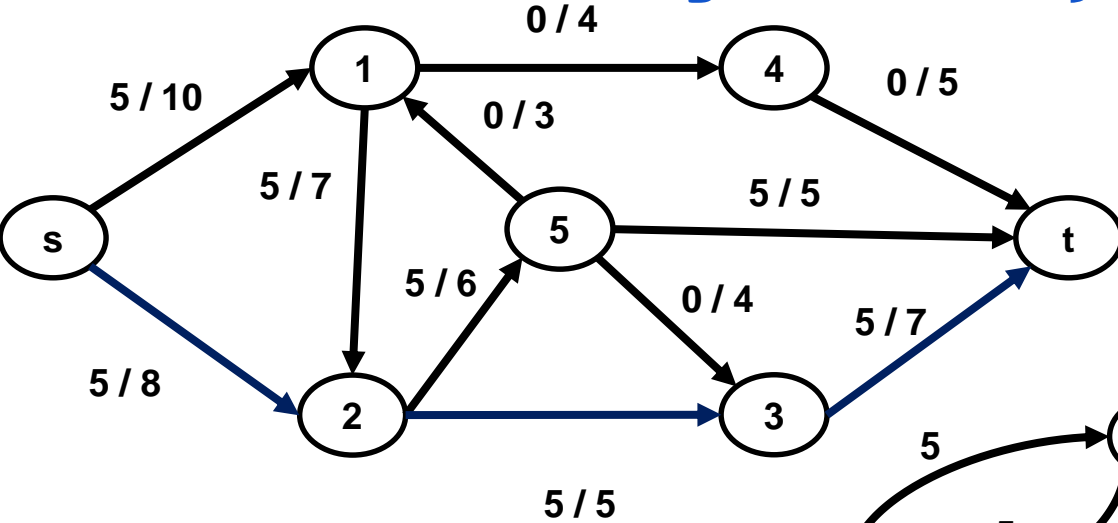
Ford-Fulkerson algorithm - Ejemplo



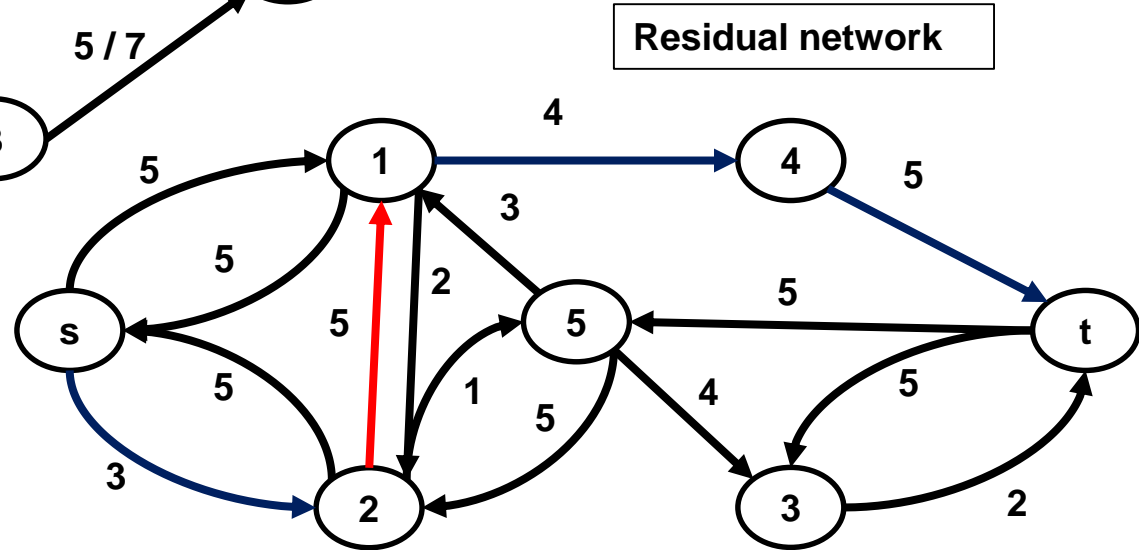
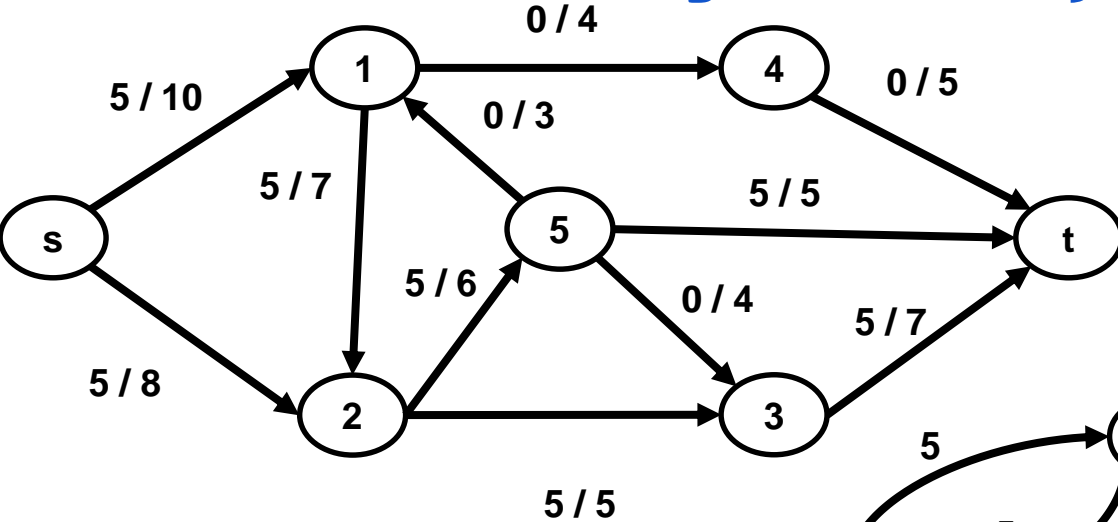
Ford-Fulkerson algorithm - Ejemplo



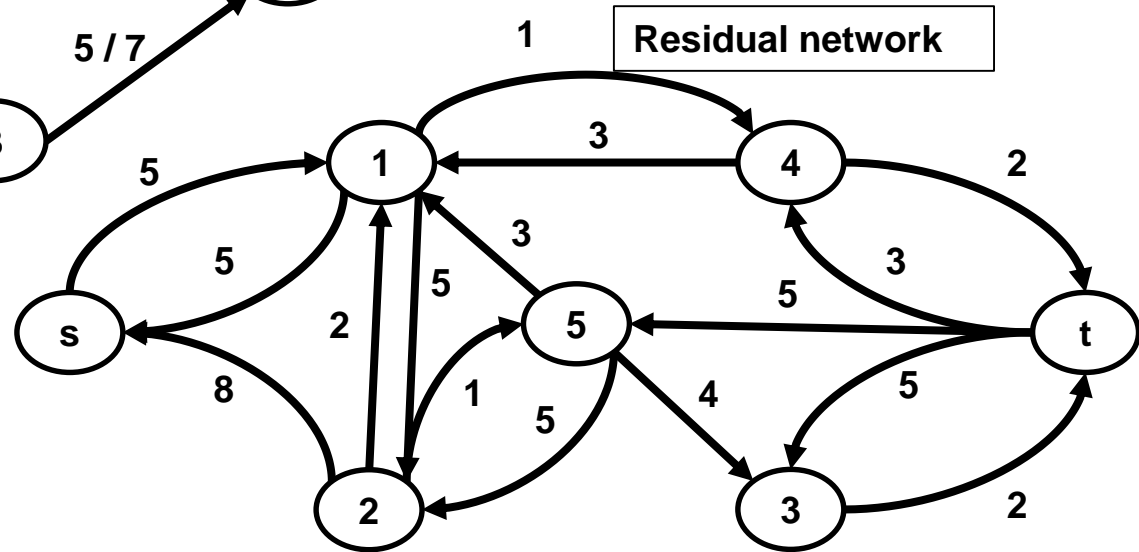
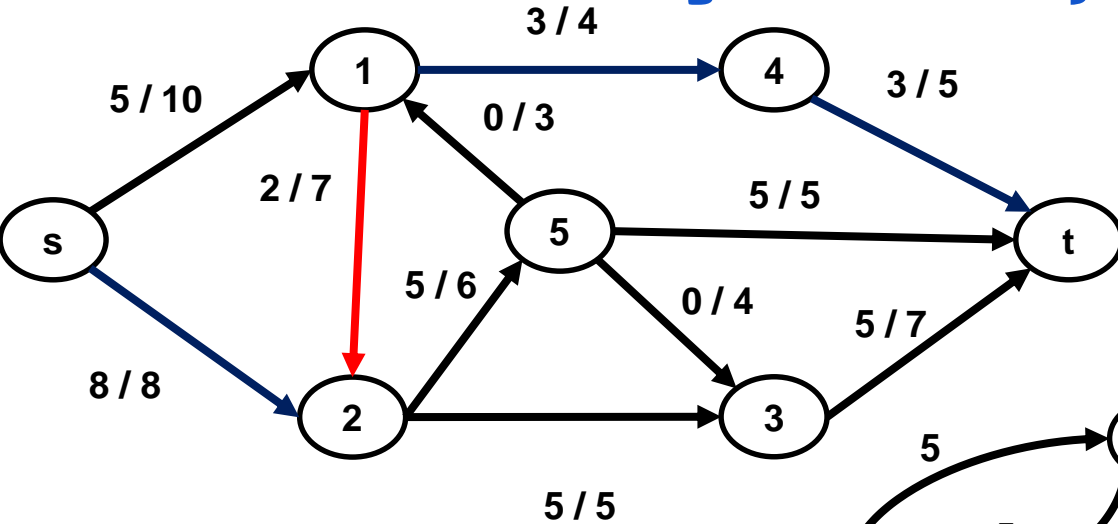
Ford-Fulkerson algorithm - Ejemplo



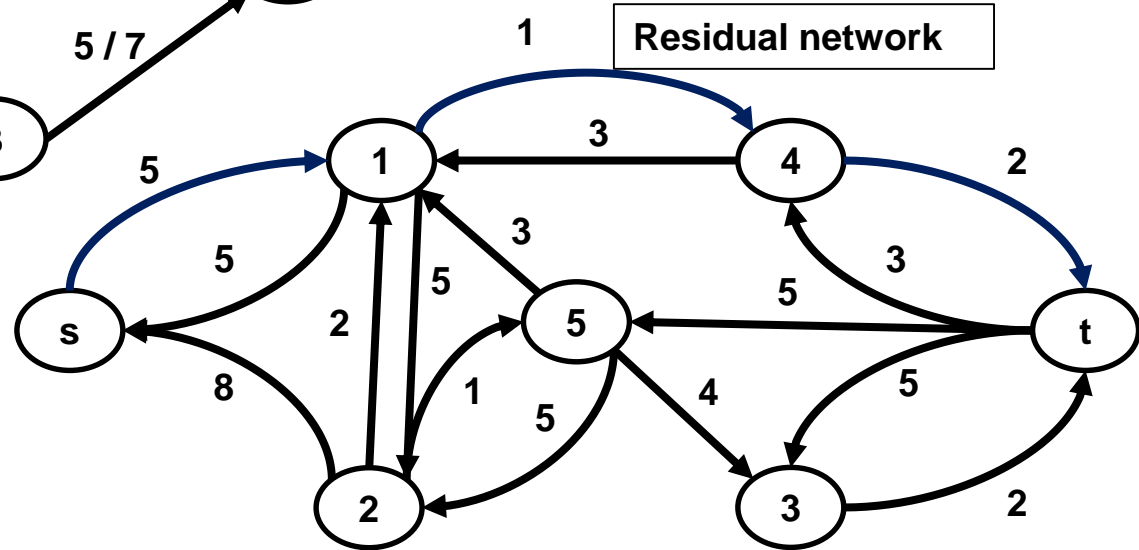
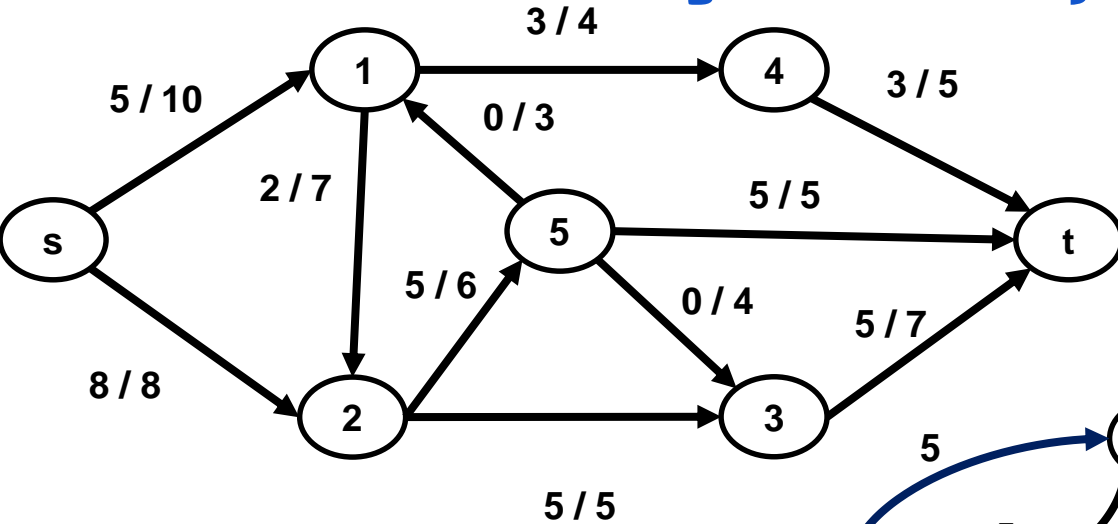
Ford-Fulkerson algorithm - Ejemplo



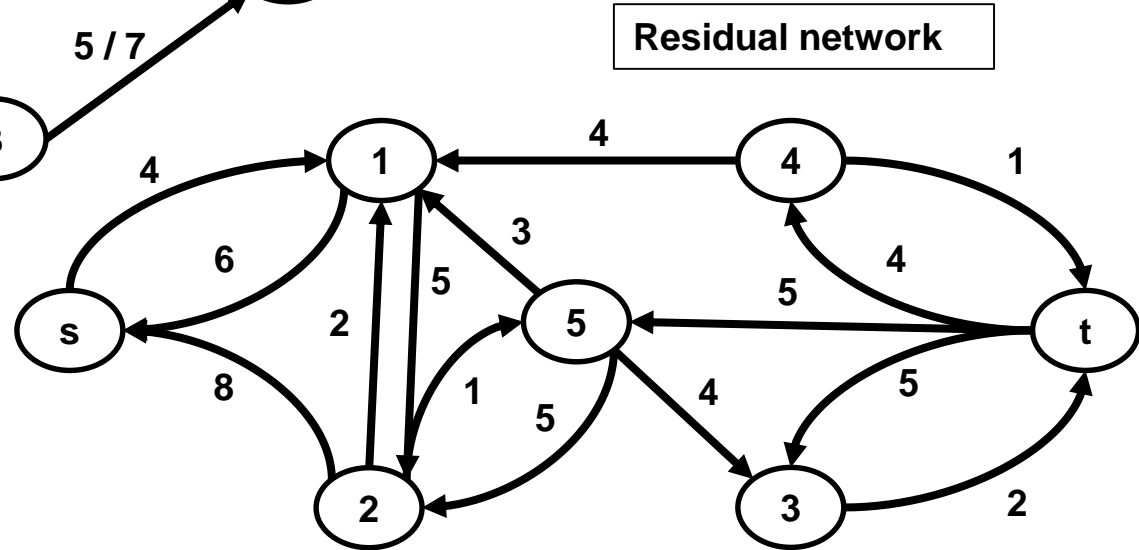
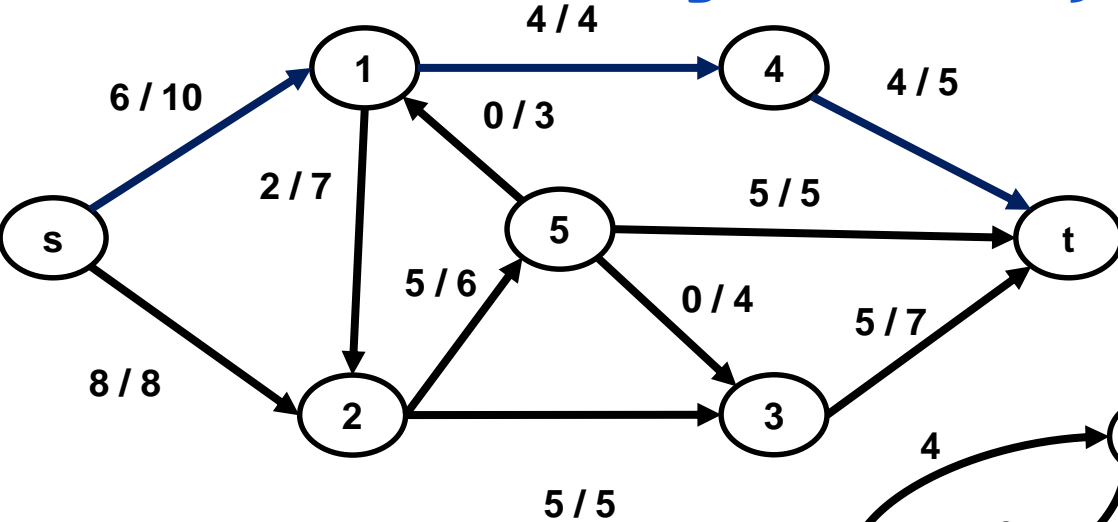
Ford-Fulkerson algorithm - Ejemplo



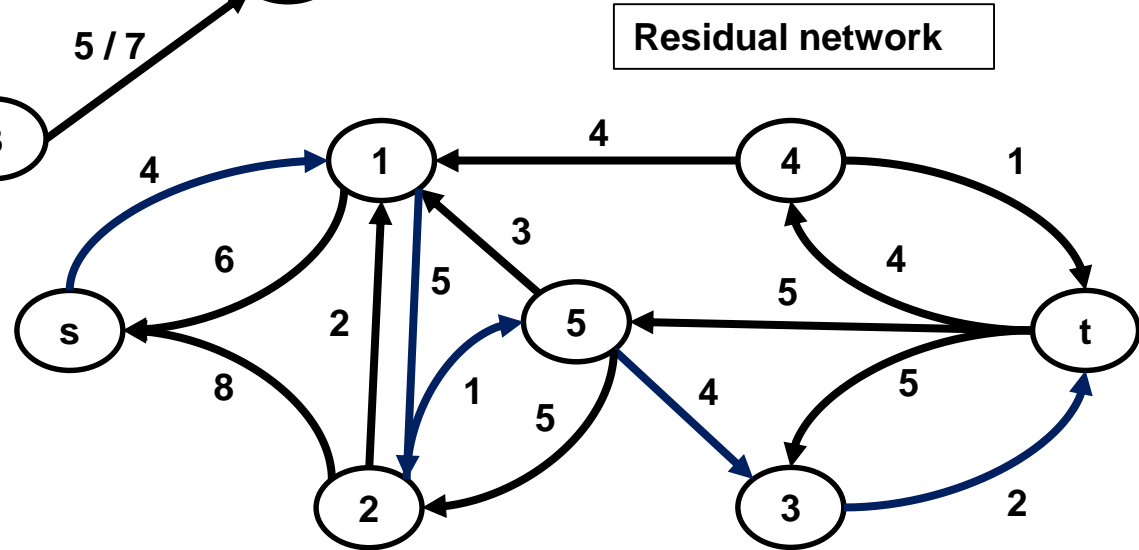
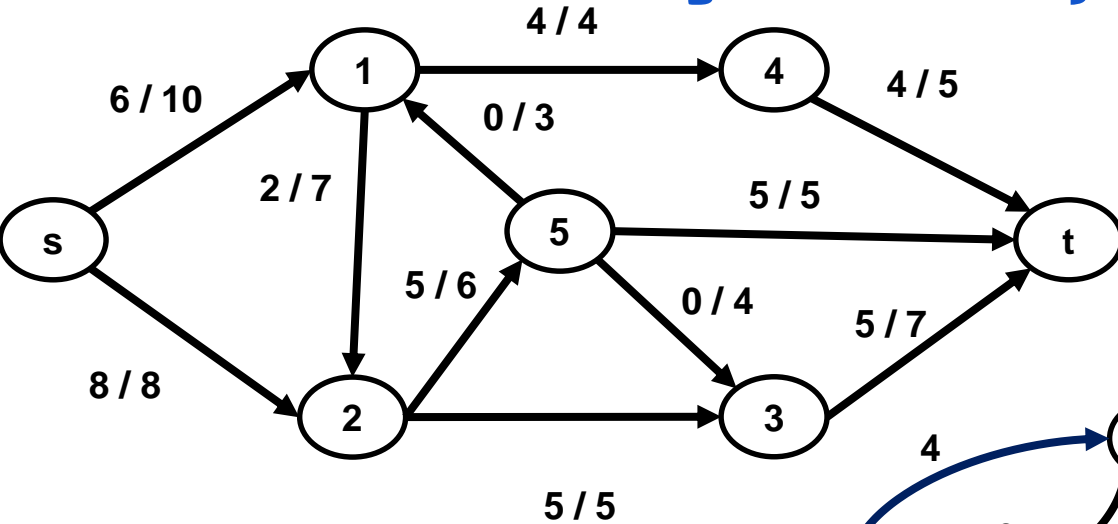
Ford-Fulkerson algorithm - Ejemplo



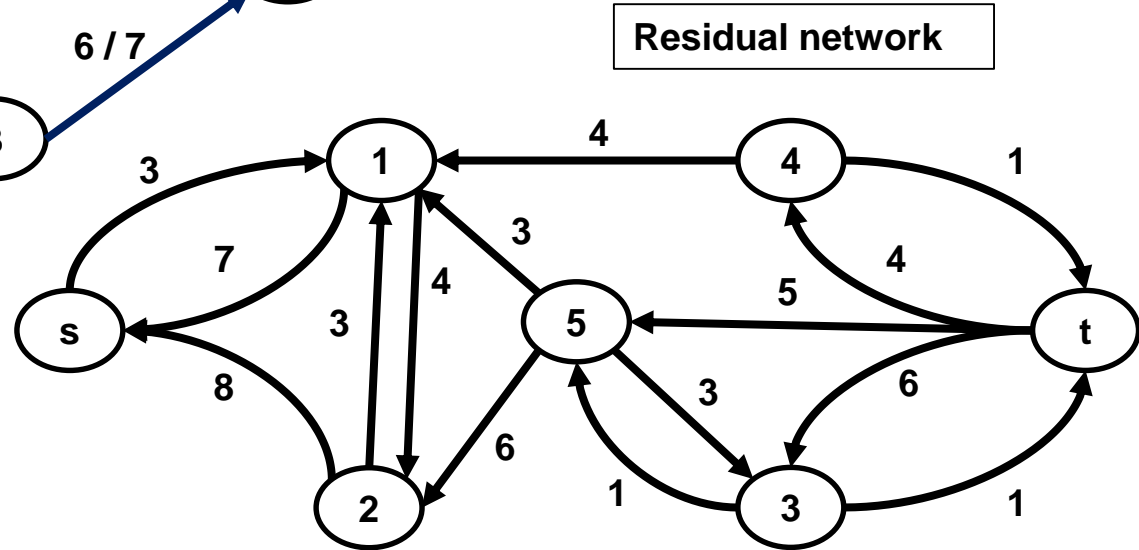
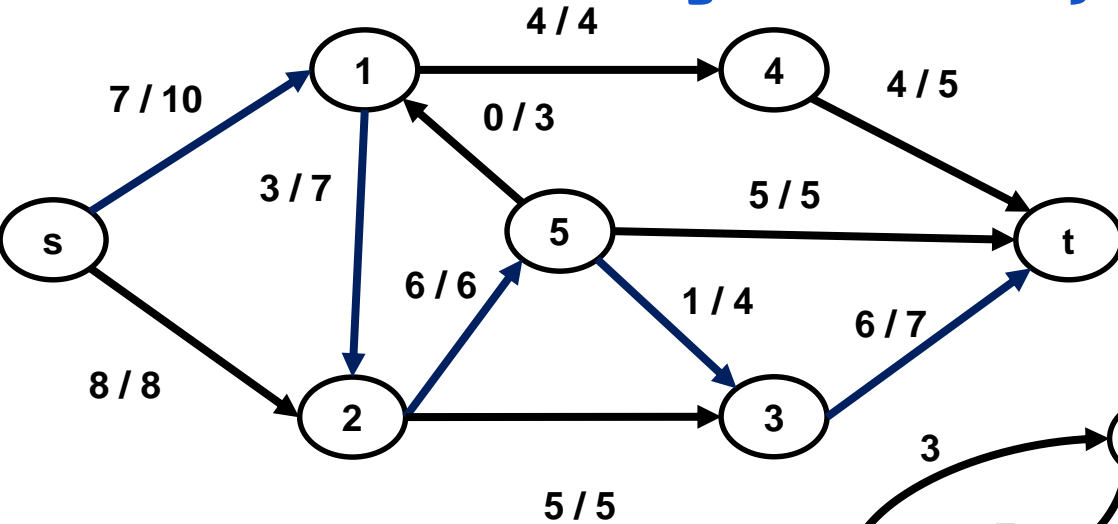
Ford-Fulkerson algorithm - Ejemplo



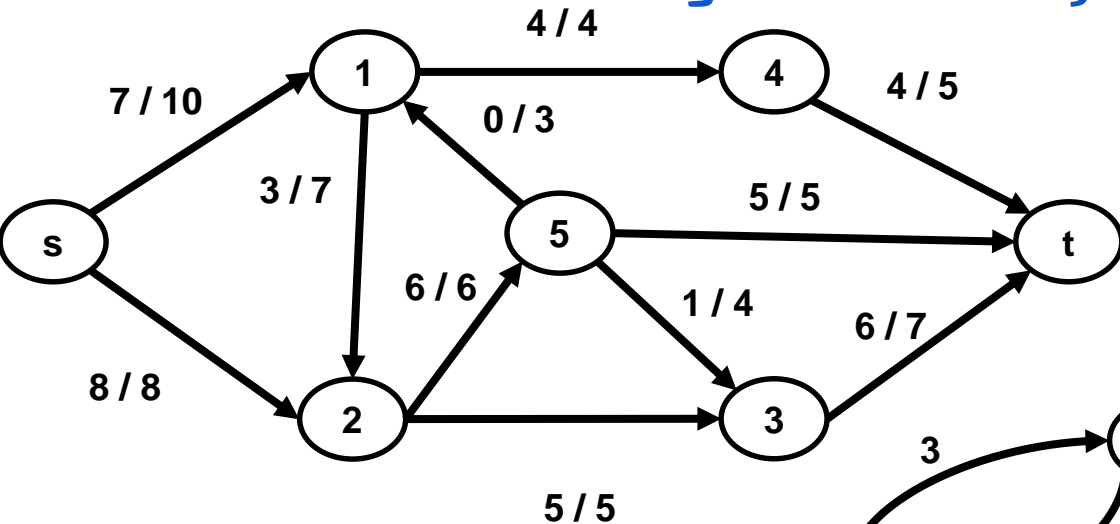
Ford-Fulkerson algorithm - Ejemplo



Ford-Fulkerson algorithm - Ejemplo

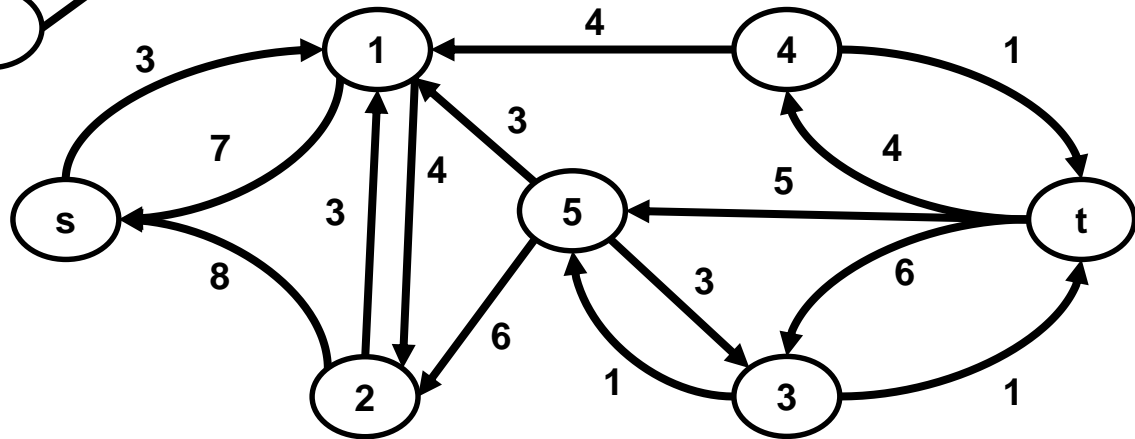


Ford-Fulkerson algorithm - Ejemplo



Max flow = 15

Residual network



Augmentation

$$(f + P)(u, v) = \begin{cases} f(u, v) + c_f(P), & (u, v) \in P \wedge (u, v) \in E \\ f(u, v) - c_f(P), & (v, u) \in P \wedge (u, v) \in E \\ f(u, v), & (u, v) \notin P \wedge (v, u) \notin P \end{cases}$$

□ **Lemma 1:** Sea f una función de flujo en G y sea P un *augmenting path* en G_f , entonces $f + P$ es también una función de flujo en G .

Demostración:

Sea $f' = f + P$ el flujo resultante

➤ Capacity constraint

Si $(u, v) \notin P, (v, u) \notin P$, entonces $f'(u, v) = f(u, v)$. Como f era un flujo válido, entonces
 $\Rightarrow 0 \leq f'(u, v) \leq c(u, v)$

Caso contrario, tenemos dos casos. O la arista $(u, v) \in E$ está en el augmenting path, o su reverso lo estaba.

❖ $(u, v) \in P$

$\Rightarrow f'(u, v) = f(u, v) + c_f(P)$ (por definición de augmentation)

Y $0 \leq c_f(P) \leq c_f(u, v)$ (por la definición de $c_f(P)$)

Lower bound

$\Rightarrow f'(u, v) \geq f(u, v) \geq 0$ (por el hecho de que $c_f(P) \geq 0$ y que f era flujo válido)

Upper bound

$\Rightarrow f'(u, v) \leq f(u, v) + c_f(u, v)$ (por el hecho de que $c_f(P) \leq c_f(u, v)$)

$\Rightarrow f'(u, v) \leq f(u, v) + c(u, v) - f(u, v)$ (por la definición de $c_f(u, v)$)

$\Rightarrow f'(u, v) \leq c(u, v)$

Augmentation

$$(f + P)(u, v) = \begin{cases} f(u, v) + c_f(P), & (u, v) \in P \wedge (u, v) \in E \\ f(u, v) - c_f(P), & (v, u) \in P \wedge (u, v) \in E \\ f(u, v), & (u, v) \notin P \wedge (v, u) \notin P \end{cases}$$

□ **Lemma 1:** Sea f una función de flujo en G y sea P un *augmenting path* en G_f , entonces $f + P$ es también una función de flujo en G .

Demostración:

Sea $f' = f + P$ el flujo resultante

➤ Capacity constraint

❖ $(v, u) \in P$ (backward edge)

$$\Rightarrow f'(u, v) = f(u, v) - c_f(P)$$

(por definición de augmentation)

$$\text{Y } 0 \leq c_f(P) \leq c_f(v, u)$$

(por la definición de $c_f(P)$)

Upper bound

$$\Rightarrow f'(u, v) \leq f(u, v) \leq c(u, v)$$

(por el hecho de que $c_f(P) \geq 0$ y que f era flujo válido)

Lower bound

$$\Rightarrow f'(u, v) \geq f(u, v) - c_f(v, u)$$

(por el hecho que $c_f(P) \leq c_f(v, u)$)

$$\Rightarrow f'(u, v) \geq f(u, v) - f(u, v)$$

(por la definición de $c_f(v, u)$)

$$\Rightarrow f'(u, v) \geq 0$$

Augmentation

$$(f + P)(u, v) = \begin{cases} f(u, v) + c_f(P), & (u, v) \in P \wedge (u, v) \in E \\ f(u, v) - c_f(P), & (v, u) \in P \wedge (u, v) \in E \\ f(u, v), & (u, v) \notin P \wedge (v, u) \notin P \end{cases}$$

□ **Lemma 1:** Sea f una función de flujo en G y sea P un *augmenting path* en G_f , entonces $f + P$ es también una función de flujo en G .

Demostración:

Sea $f' = f + P$ el flujo resultante. Sea $net'(u)$ el flujo neto resultante en cada nodo $u \in V$

➤ Flow conservation

Como P es un simple path, cada nodo aparece a lo más una vez en el path. Solo los nodos que aparecen en P tendrán alteraciones en su flujo neto, por lo que $\forall u \notin P, net'(u) = 0$.

Luego, $\forall u \in P - \{s, t\}$ tendremos una arista saliente y una arista entrante (**ambas aristas en G_f**).

Analicemos el cambio en el flujo neto para el nodo u dependiente del delta por la arista entrante y saliente.

$$net'(u) = net(u) + \Delta out(u) + \Delta in(u)$$

❖ Para la arista saliente $(u, out) \in P$

Si $(u, out) \in E \Rightarrow f'(u, out) = f(u, out) + c_f(P) \Rightarrow \Delta out(u) = c_f(P)$ (aumentó una arista entrante en G)

Si $(u, out) \notin E \Rightarrow f'(out, u) = f(out, u) - c_f(P) \Rightarrow \Delta out(u) = c_f(P)$ (disminuyó una arista saliente en G)

❖ Para la arista entrante $(in, u) \in P$

Si $(in, u) \in E \Rightarrow f'(in, u) = f(in, u) + c_f(P) \Rightarrow \Delta in(u) = -c_f(P)$ (aumentó una arista entrante en G)

Si $(in, u) \notin E \Rightarrow f'(u, in) = f(u, in) - c_f(P) \Rightarrow \Delta in(u) = -c_f(P)$ (disminuyó una arista saliente en G)

Entonces $net'(u) = net(u) + c_f(P) - c_f(P) = net(u) = 0$ debido a que f era flujo válido ■

Augmentation

$$(f + P)(u, v) = \begin{cases} f(u, v) + c_f(P), & (u, v) \in P \wedge (u, v) \in E \\ f(u, v) - c_f(P), & (v, u) \in P \wedge (u, v) \in E \\ f(u, v), & (u, v) \notin P \wedge (v, u) \notin P \end{cases}$$

□ **Lemma 2:** Sea f una función de flujo en G y sea P un *augmenting path* en G_f , entonces el valor del nuevo flujo $|f + P| = |f| + c_f(P)$

Demostración:

Solo necesitamos hallar el flujo neto en s . Nota que s solo tiene una arista saliente en P (ojo que es una arista saliente en el grafo residual G_f). Aplicamos la misma lógica que en el lema 1:

$$net'(s) = net(s) + \Delta out(s)$$

Para la arista saliente $(s, out) \in P$

Si $(s, out) \in E \Rightarrow f'(s, out) = f(s, out) + c_f(P) \Rightarrow \Delta out(s) = c_f(P)$ (aumentó una arista entrante en G)

Si $(s, out) \notin E \Rightarrow f'(out, s) = f(out, s) - c_f(P) \Rightarrow \Delta out(s) = c_f(P)$ (disminuyó una arista saliente en G)





$$\Rightarrow net'(s) = net(s) + c_f(P) = |f| + c_f(P) \quad \blacksquare$$

□ **Corolario 3:** Sea f una función de flujo en G . Si existen augmenting paths en G_f , entonces f no es max flow.

Demostración:

Por el lemma 2, podemos construir un nuevo flujo $f' = f + P$ cuyo valor es $|f| + c_f(P)$. Como $c_f(P) > 0$ por definición, entonces $|f'| > |f|$ y f no es max flow. ■

Contenido

1. Flows	
2. Residual Networks	
3. Cuts	
4. Ford-Fulkerson algorithm	

Simple bound of a max flow.

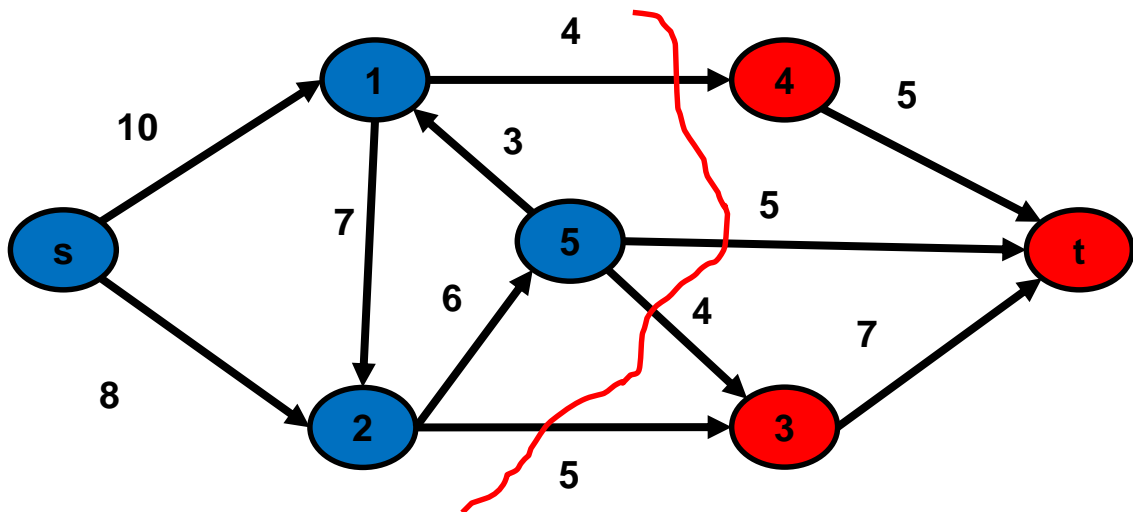
- Si queremos demostrar la optimalidad del algoritmo de Ford-Fulkerson, empecemos por encontrar algunos *upper bound* del max flow.

$$\begin{aligned}|f| = net(s) &= \sum_{out} f(s, out) - \sum_{in} f(in, s) \\ \Rightarrow |f| &\leq \sum_{out} f(s, out) \\ \Rightarrow |f| &\leq \sum_{out} c(s, out)\end{aligned}$$

Cut

- ❑ Para ir más lejos, tenemos que generalizar un poco más nuestro *bound* anterior.
- ❑ **Cut:** Un corte (S, T) es una partición de V en dos subconjuntos S y $T = V - S$ tal que $s \in S$ y $t \in T$. También se define la capacidad de un *cut* como la suma de capacidades de las aristas que empiezan en S y terminan en T .

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$



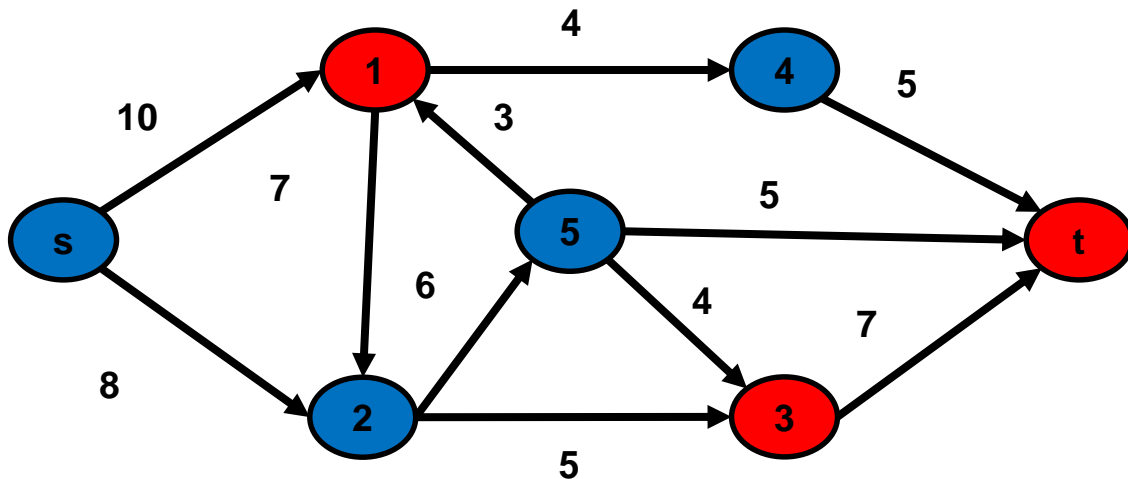
En el ejemplo de la izquierda tenemos un corte con el conjunto S de nodos azules y T de nodos rojos.

La capacidad del corte es
 $4 + 5 + 4 + 5 = 18$

Ten en cuenta que solo importan las aristas de S a T y no las de T a S

Cut

- ❑ La definición no requiere que los subconjuntos estén “conectados”. Por ejemplo, tenemos tener un corte como el siguiente:



En el ejemplo de la izquierda tenemos un corte con el conjunto S de nodos azules y T de nodos rojos.

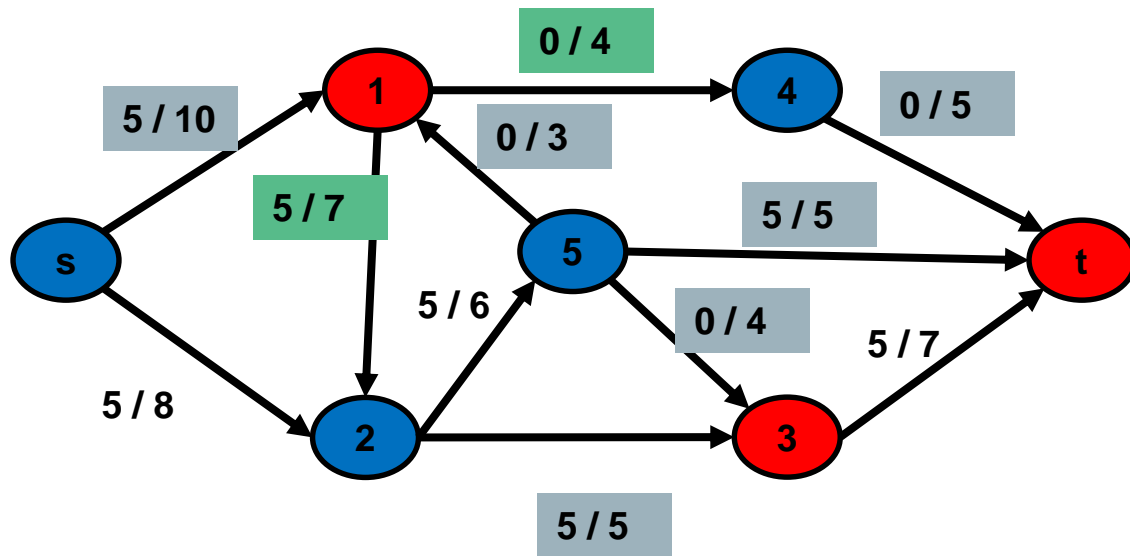
La capacidad del corte es
 $10 + 3 + 5 + 5 + 4 + 5 = 32$

- ❑ Otro problema clásico es hallar el mínimo corte (el que tenga menor capacidad).

Flujo a través de un corte

□ **Definición:** Sea f un flujo válido en G . Se define al flujo a través de un corte (S, T) como el flujo que va de S a T menos el flujo que va de T a S y se denota como $f(S, T)$.

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$



$$f(S, T) = (5 + 0 + 0 + 5 + 0 + 5)$$

$$-(5 + 0)$$

$$= 10$$

Flujo a través de un corte

□ **Lemma 4:** Sea f un flujo válido en G y sea (S, T) cualquier corte en G .
 $|f| = f(S, T)$

Demostración:

Por *flow conservation*:

$$\begin{aligned}\forall u \in V - \{s, t\}, \quad net(u) &= 0 \\ \Rightarrow \sum_{u \in S} net(u) &= net(s) = |f| \\ \Rightarrow |f| &= \sum_{u \in S} \left(\sum_{out \in V} f(u, out) - \sum_{in \in V} f(in, u) \right)\end{aligned}$$

Como sabemos que $V = S \cup T$ y $S \cap T = \emptyset$ entonces podemos dividir V en la sumatoria

$$\begin{aligned}\Rightarrow |f| &= \sum_{u \in S} \sum_{out \in S} f(u, out) - \sum_{u \in S} \sum_{in \in S} f(in, u) + \sum_{u \in S} \sum_{out \in T} f(u, out) - \sum_{u \in S} \sum_{in \in T} f(in, u) \\ \Rightarrow |f| &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) = f(S, T) \blacksquare\end{aligned}$$

Better bound for max flow

□ **Corolario 5:** Sea f un flujo válido en G y sea (S, T) cualquier corte en G .
 $|f| \leq c(S, T)$

Demostración:

Usando el lemma 4

$$\begin{aligned} |f| = f(S, T) &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) \\ |f| &\leq \sum_{u \in S} \sum_{v \in T} f(u, v) && \text{(debido a que } f \text{ era flujo válido y es no negativo)} \\ |f| &\leq \sum_{u \in S} \sum_{v \in T} c(u, v) && \text{(usando } \textit{capacity constraints}) \\ |f| &\leq c(S, T) \quad \blacksquare \end{aligned}$$

Del corolario anterior se deduce que $\max flow \leq \min cut$. Es más, si encontramos un flujo y un corte que tengan el mismo valor, entonces habremos encontrado el max flow.

Contenido

1. Flows	
2. Residual Networks	
3. Cuts	
4. Ford-Fulkerson algorithm	

Max flow-Min Cut Theorem

□ **Theorem 6 (Max flow – Min Cut Theorem):** Si f es un $(s - t)$ flow en $G(V, E)$ y no existen augmenting paths en G_f , entonces existe un cut (S, T) tal que $|f| = c(S, T)$. Más aún, f es un max flow y (S, T) es un min cut.

Demostración:

Si no existen augmenting paths, podemos construir un corte (S, T) de la siguiente forma:

$S = \{u \in V \text{ y existe un path de } s \text{ a } u \text{ en } G_f\}$, $T = V - S$.

Esta partición es un corte debido a que $s \in S$ (trivial) y $t \notin S$ porque no existen augmenting paths.

Para todo par de vértices (u, v) tal que $u \in S$ y $v \in T$, tenemos que $c_f(u, v) = 0$, ya que, de lo contrario, la arista (u, v) existiría en el residual G_f lo cual implicaría que $v \notin T$ (contradicción). Aquí tenemos tres casos.

- ❖ Si $(u, v) \in E$, entonces $c_f(u, v) = c(u, v) - f(u, v) \Rightarrow f(u, v) = c(u, v)$
- ❖ Si $(v, u) \in E$, entonces $c_f(u, v) = f(v, u) \Rightarrow f(v, u) = 0$
- ❖ En cualquier otro caso, significa que $f(u, v) = f(v, u) = 0$ porque no existía ni la arista directa ni reversa.

$$\Rightarrow f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) = \sum_{u \in S} \sum_{v \in T} c(u, v) - \sum_{u \in S} \sum_{v \in T} 0 = c(S, T)$$

Por el lemma 4 tenemos que $|f| = f(S, T)$ y entonces $|f| = c(S, T)$. Finalmente, por el corolario 5 tenemos que f es max flow y (S, T) es min cut ■

Volviendo a Ford-Fulkerson

- Recordemos que el algoritmo de Ford-Fulkerson intenta hacer augmentations hasta que ya no existan augmenting paths.

Algorithm 2: Ford-Fulkerson algorithm

```
input :  $G(V, E, c)$  , source :  $s$ , sink :  $t$   
output: max flow:  $f^*$   
1 Initialize flow  $f^*$  to 0  
2 while there exists a path  $P$  in the residual network  $G_f$  from  $s$  to  $t$  do  
3    $c_f(P) = \min_{(u,v) \in P} c_f(u, v)$   
4   foreach edge  $(u, v) \in P$  do  
5     if  $(u, v) \in E$  then  
6        $f(u, v) = f(u, v) + c_f(P)$   
7     else  
8        $f(v, u) = f(v, u) - c_f(P)$   
9     end  
10  end  
11 end  
12 return  $f^*$ 
```

Correctness Ford-Fulkerson

□ **Lemma 7:** Cuando el algoritmo de Ford-Fulkerson termina, el resultado es una función de flujo válida y es un max flow.

Demostración:

- ✓ **El flujo resultante es válido.** Por el lemma 1 sabemos que cada *augmentation* produce una función de flujo válida y por el corolario 3 sabemos que si existió un augmenting path, entonces el flujo f no era máximo.
- ✓ **El flujo resultante es máximo.** Por el max flow – min cut theorem sabemos que como al final del algoritmo ya no hay augmenting paths, entonces el flujo es máximo.

Min cut Algorithm

La demostración del **Max flow – Min Cut Theorem** incluso nos provee una forma de encontrar un corte mínimo.

Por el lemma 7, sabemos que Ford-Fulkerson es correcto por lo que al terminar el algoritmo, hemos encontrado un flujo máximo. Por el corolario 3, al tener un flujo máximo, ya no hay augmenting paths.

Utilizando la demostración del **Max flow – Min Cut Theorem**, podemos construir un corte mínimo de esta forma:

$$S = \{u \in V \text{ y existe un path de } s \text{ a } u \text{ en } G_f\}, T = V - S.$$

Es decir, basta con hacer un *dfs* o *bfs* desde s en la **red residual**. Todos los nodos alcanzados serán parte de S y el resto de T .

Integrality Theorem

□ **Teorema 8 (Integrality Theorem):** Si todas las capacidades de la red original G son enteras, entonces existe un flujo máximo donde el flujo sobre cada arista sea entero.

Demostración:

Utilizando el algoritmo de Ford-Fulkerson por inducción

- **Caso base:** Al principio todos los flujos son 0, un entero.
- **Paso inductivo:** Por la hipótesis inductiva, si luego de x iteraciones $f(u, v)$ es entero para toda arista. En caso ya no haya augmenting paths, entonces hemos terminado. Caso contrario, se hará un augmentation donde algunas aristas se les sumará o restará $c_f(P)$. Como todos los flujos son enteros, las capacidades residuales $c_f(u, v) = c(u, v) - f(u, v)$ también lo son, lo cual implica que el mínimo entre ciertas capacidades residuales también es entero, lo que indica que $c_f(P)$ es entero. Finalmente, como los flujos de las aristas alteradas se les sumará o restará un entero, eso quiere decir que permanecerán enteros en la iteración $x + 1$ ■

Time Complexity — Ford Fulkerson

□ **Lemma 9:** Si todas las capacidades de la red original G son enteras, el algoritmo de Ford-Fulkerson terminará.

Demostración:

Por el *Integrality Theorem* y su demostración, sabemos que la red permanecerá con flujos enteros en cada momento.

Por el lemma 2, sabemos que en cada iteración, el valor del flujo $|f|$ aumenta en $c_f(P)$. Sabemos que $c_f(P) > 0$ y que es entero. Por lo tanto $c_f(P) \geq 1$.

En cada iteración el valor del flujo aumenta en al menos 1 unidad. Como el valor del flujo es finito y está acotado por

$$\sum_u c(s, u)$$

Entonces el algoritmo de Ford-Fulkerson terminará en luego de un número finito de iteraciones. ■

Time Complexity — Ford Fulkerson

- ❑ Para hallar un augmenting path podemos usar cualquier búsqueda como dfs o bfs en $O(E')$ donde E' son las aristas en el grafo residual G_f . Sin embargo, por cada arista del grafo original, a lo mucho hay 2 aristas en el residual (directa y reversa) ya que hemos asumido que no hay anti-parallel edges. Por lo tanto $E \leq 2E'$.
- ❑ Si denotamos al flujo máximo como F , entonces para un grafo con **capacidades enteras**, en el peor caso cada augmentation aumenta solo 1 el valor de flujo. A lo más habrán $O(F)$ iteraciones. En cada iteración hacemos una búsqueda en $O(E') = O(E)$ y un augmentation en $O(V)$ por lo que la complejidad del algoritmo es $O(EF)$.

Recordemos que $F \leq \sum_u c(s, u)$. Si además denotamos U como la máxima capacidad, entonces $F \leq E \times U$. Entonces la complejidad también puede ser denotado como $O(E^2 \times U)$



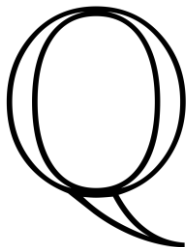
Time Complexity — Ford Fulkerson

□ ¿Y si fueran capacidades **racionales**?

Supongamos que $c(u, v) = \frac{P(u, v)}{Q(u, v)}$ con P y Q como funciones enteras. Entonces podemos “homogenizar” todas las fracciones transformándolas a un denominador que sea el LCM de todos los denominadores. Sea $L = \text{lcm}(\{P(u, v), \forall(u, v) \in E\})$. Entonces podemos expresar $c(u, v) = \frac{P'(u, v)}{L}$.

Por inducción es fácil notar que en cada iteración del algoritmo podemos seguir expresando cada capacidad residual como $\frac{x}{L}$ y el valor del flujo final también se puede expresar con un denominador L .

Por lo tanto, podemos deducir que si corremos el algoritmo con las capacidades racionales, en el peor caso aumentaría el valor de flujo en $\frac{1}{L}$. Si el flujo máximo tiene valor F , entonces el algoritmo terminaría en $O(FL)$ iteraciones, siendo la complejidad total $O(EFL)$.

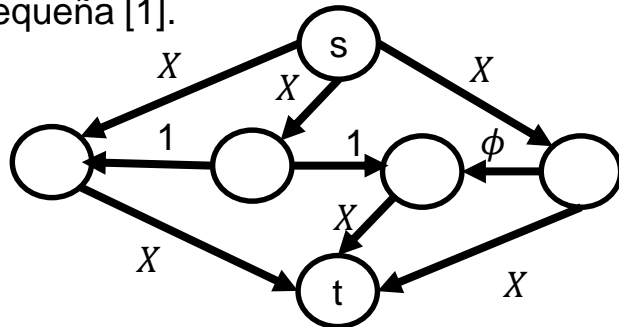


Time Complexity – Ford Fulkerson

- ❑ ¿Y si fueran capacidades **irracionales**?
- ❑ ¿Cómo podríamos estar seguros de que cada aumento no se hace más pequeño que el anterior y que se podrían necesitar un número infinito de augmentations para terminar?

Desafortunadamente con capacidades irracionales el algoritmo de Ford Fulkerson podría no terminar si tiene la mala suerte de tomar malas elecciones. Es incluso peor, puede ser que incluso cuando tiende al infinito, el algoritmo no converja a un max flow.

Ford y Fulkerson describieron en 1962 una red de flujo que exhibía este comportamiento. Más tarde, Uri Zwick encontraría una red más pequeña [1].



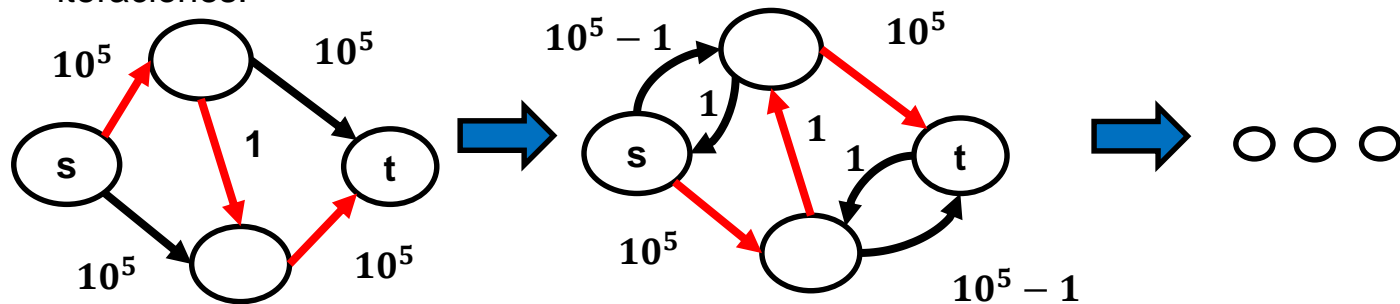
En el gráfico X puede ser un número grande y ϕ es el número áureo

Fuente: Adaptado de [1]

[1] Jeff Erickson. Algorithms. Chapter 10: Maximum Flows and Minimum cuts.

Time Complexity – Ford Fulkerson

- ❑ Afortunadamente, de forma práctica las capacidades irracionales no nos molestan mucho, porque es imposible para una computadora representar de forma correcta un número irracional con sus infinitos decimales.
- ❑ Además, la mayoría de veces trabajaremos con enteros en los problemas de competencia. ¿Pero qué tan certera es el upper bound de $O(EF)$? ¿O al menos que haga $O(F)$ augmentations?
- ❑ Teóricamente si uno no elige bien los *augmenting path* podemos crear casos que lleguen a $O(F)$ iteraciones.



- ❑ Sin embargo, en la práctica es más complicado hackear alguna solución que use por ejemplo *dfs*, ya que el grafo va a estar cambiando mucho en cada iteración. Se vuelve incluso mucho más difícil llegar a la cota límite cuando el grafo no viene como input explícitamente sino que es construido a partir de otro input.

Implementation details

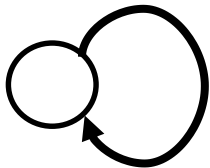
Para representar al flujo y la capacidad podemos usar matrices de dos dimensiones $c_{n \times n}$ y $f_{n \times n}$. Sin embargo, tenemos algunos inconvenientes que debemos resolver.

- ❑ **Multiple edges – Self loops:** ¿Cómo generalizar el algoritmo cuando hay múltiples aristas o aristas entre un mismo nodo?
- ❑ **Not implementation friendly:** La implementación del algoritmo tal cual como se mostró puede ser tediosa a la hora del augmentation ya que tenemos que distinguir entre aristas originales y aristas reversas para poder realizar la operación. Además necesitamos un grafo residual.
- ❑ **Antiparallel edges:** Algo que asumimos al principio fue que no habría antiparallel edges. ¿cómo generalizar el algoritmo para soportar antiparallel edges?

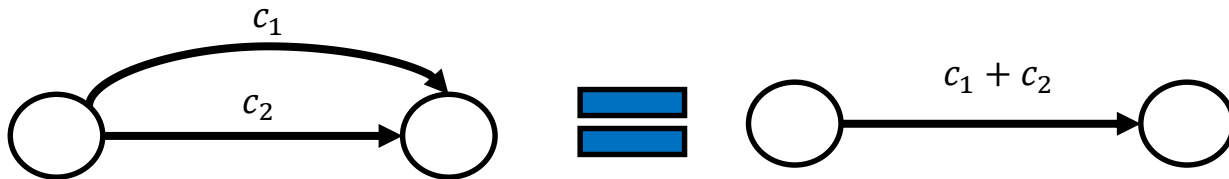
Implementation details

❑ **Multiple edges – Self loops:** ¿Cómo generalizar el algoritmo cuando hay múltiples aristas?

Así como los ciclos, los self loops no importan en lo absoluto. Es fácil demostrarlo, porque si tomamos cualquier flujo máximo que utilice algún self loops, podemos llevar su flujo a 0 y esto no va a alterar el flujo neto, por lo que el flow conservation se mantiene y las capacity constraints se siguen cumpliendo. En conclusión podemos ignorarlos.



En el caso de multi-edges, estas a priori podrían representar un problema ya que $c(u, v)$ ya no estaría bien definido, sino que ahora se deberían trabajar las aristas como objetos para poder diferenciarse $c(e)$. Sin embargo, todavía podemos salvarlo haciendo la siguiente jugada.



Implementation details

❑ Not implementation friendly:

En primer lugar, no necesitamos construir explícitamente el grafo residual, podemos tener implícitamente las capacidades residuales si es que siempre guardamos la capacidad de cada arista y el flujo actual que pasa por ella.

Luego, para hacer más fácil la implementación, podemos notar que en el algoritmo solo importan las capacidades residuales para tomar las decisiones. Y podemos hacer fácilmente que la capacidad residual suba o baje si es que cambiamos un poco la definición de un flujo con estas nuevas condiciones.

- **Capacity Constraint:** $f(u, v) \leq c(u, v)$ (el flujo puede ser negativo), $\forall u, v \in V$
- **Skew Symmetry:** $f(u, v) = -f(v, u)$, $\forall u, v \in V$
- **Flow conservation:** $\forall u \in V - \{s, t\}$ $\sum_{out} f(u, out) = 0$

Utilizando esta nueva definición, la capacidad ahora siempre será $c_f(u, v) = c(u, v) - f(u, v)$, sin importar si es arista original o reversa. Esto es porque cuando es una arista reversa tendremos $c_f(u, v) = 0 - f(u, v) = f(v, u)$, que corresponde a lo que teníamos definido antes.

Tener en cuenta que ahora en cada augmentation debemos actualizar la arista directa y reversa para mantener el skew symmetry.

Implementation details

Algorithm 3: Ford-Fulkerson improved

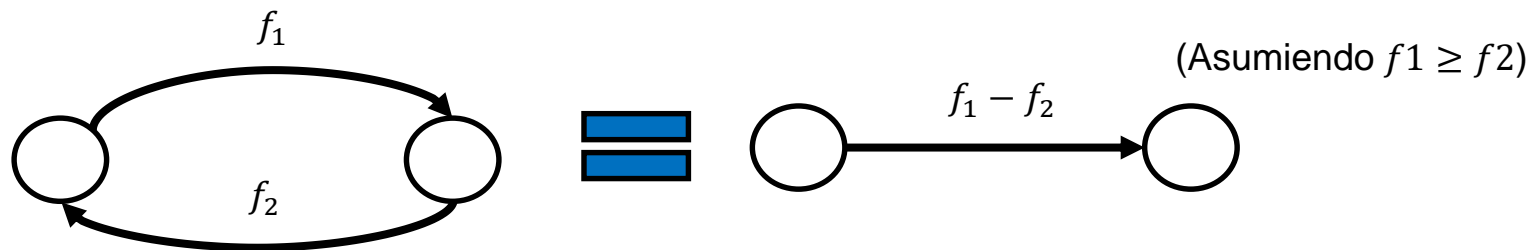
input : $G(V, E, c)$, *source* : s , *sink* : t
output: max flow: f^*

- 1 Initialize flow f^* to 0
- 2 **while** *there exists a path P in the residual network G_f from s to t* **do**
- 3 $c_f(P) = \min_{(u,v) \in P} c_f(u, v)$
- 4 **foreach** *edge* $(u, v) \in P$ **do**
- 5 $f(u, v) = f(u, v) + c_f(P)$
- 6 $f(v, u) = f(v, u) - c_f(P)$
- 7 **end**
- 8 **end**
- 9 **return** f^*

Implementation details

□ Antiparallel edges:

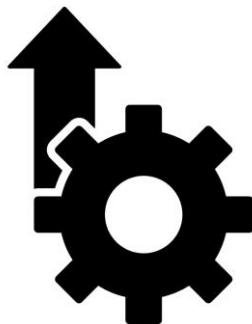
Notemos que por nuestra observación inicial de que los ciclos son inútiles, no tiene sentido mandar flujo en ambas direcciones de una arista. Es decir



Por otro lado, utilizando el **skew symmetry** de la diapo anterior, como ya no tenemos que preocuparnos por quién es arista original o reversa, ya podemos permitir los antiparalel edges. Podemos poner capacidad positiva en ambas direcciones de una arista, pero solo por una de ellas pasará flujo, mientras que por la otra estará el negativo del flujo actual.

Improvement on memory

Finalmente, recordemos que nuestro algoritmo utilizaba $O(V^2)$ de memoria. Sin embargo, cuando la cantidad de nodos es muy grande, podemos reducirlo solo a $O(E)$ de memoria utilizando la lista de adyacencia que siempre usamos. Sin embargo, deberemos utilizar la noción de *multi-graph* permitiendo que existan multiple edges sin combinarlas. Para cada arista le crearemos una arista reversa en la lista de adyacencia y deberemos guardar su posición en la lista.



Referencias

- ❑ [1] Jeff Erickson. Algorithms. Chapter 10: Maximum Flows and Minimum cuts.
<https://jeffe.cs.illinois.edu/teaching/algorithms/book/10-maxflow.pdf>
- ❑ [2] cp-algorithms. Maximum Flow. https://cp-algorithms.com/graph/edmonds_karp.html
- ❑ [3] Kleinberg and Tardos, Algorithm Design.
- ❑ [4] CLRS. Introduction to Algorithms. Chapter 26: Maximum Flow.